MIT/LCS/TR-355

DATA STRUCTURE MANAGEMENT IN A DATA FLOW COMPUTER SYSTEM

Bhaskar Guharoy

May 1985

Data Structure Management in a Data Flow Computer System

Bhaskar Guharoy

May 1985

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Laboratory for Computer Science
Cambridge, MA 02139

Abstract

DATA STRUCTURE MANAGEMENT
IN A
DATA FLOW COMPUTER SYSTEM
by
Bhaskar Guharoy

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 1985 in partial fulfillment of the requirements
for the Degree of Master of Science

VIM is an experimental computer system being developed at MIT for supporting functional programming. The execution mechanism of the computer is based on data flow. This thesis presents mechanisms for managing data structures in this system. The thesis also develops a methodology for designing computers, which is based on successive refinement of formal models of the computer.

A formal model L1 of the abstract architecture of VIM is first developed. The behaviour of this model is described by its operational semantics; L1 is the specification of VIM. L1 is then refined to model hierarchical physical storage consisting of main memory and disk. This refined model is called L2. The unit of storage allocation and of data transfer between main store and disk is a chunk. The thesis proposes a new data structure called VIM-tree which is a tree of chunks. Data structures in VIM (arrays and records) are stored on VIM-trees. VIM-trees allow efficient applicative operations on data structures and permit a large amount of sharing. A reference count mechanism is proposed to perform automatic storage reclamation. Special care is taken to handle operations in L2 on data structures containing *early-completion queues* and *suspensions*, which are distinctive features of VIM. A base language for this machine is outlined in the thesis.

The models L1 and L2 are then shown to be equivalent for the proposed base language. The equivalence is proved by exhibiting a McGowan mapping between the states of the two models during the execution of a program writen in the base language.

Thesis Supervisor : Jack B. Dennis

Professor of Elecrical Engineering and Computer Science

Keywords: - VIM, Dataflow Graphs, Functional Languages, Structure Management, Hierarchical Storage, Heaps, Tree structures, Early-Completion Structures, Suspensions, Streams, Machine Equivalence, VIMVAL Compilation.

Acknowledgments

There are many people who contributed, direetly or indireetly, to the writing of this thesis. I wish to thank Jaek Dennis for providing eneouragement and support during the long period of gestation of this thesis. Thanks to Suresh Jagannathan for being ever ready to listen to my ideas and eoming up with eonstructive critieism. Mueh of the formal model L1 was developed jointly with Suresh.

My sineere thanks to David Culler for being sueh a wonderful friend. Thanks to Sara Mayeno for allowing me to cook for her whenever I visited her and David. Steve Heller provided mueh of the fun and laughter. My thanks to Keshav Pingali for the many interesting diseussions I had with him. Thanks to Andy Boughton, Guang-Rong Gao, Vinod Kathail, Greg Papadopoulos, Earl Waldin and all the other members of CSG and FLA who make the laboratory such a fun place to work in.

My family has always been very encouraging and I eherish their enthusiasm and interest in my work. Finally, my thanks to Marcela for making me finally go erazy, in the right way, and for being the perfect partner in goofiness.

The woods are lovely, dark and deep
But I have promises to keep
And miles to go before I sleep
And miles to go before I sleep.

- "Stopping by woods on a snowy evening"
Robert Frost.

To Mo and [illegible]

Table of Contents

Chapter One

Introduction

In recent years data flow computer systems have been the focus of vigorous research, especially in the context of high speed scientific computations. In addition to higher speed, the data flow model of computation appears to provide a more robust programming environment than is available on conventional systems. The VIM project of the Computation Structures Group at MIT is aimed at examining the issues involved in implementing a modern, general-purpose computing environment based on the principles of dataflow that can effectively support such diverse computational applications as database systems, logic programming, etc. The ideas about the VIM system have evolved over the years, drawing much from the works of Dennis [9, 10, 11], Patil [31] and Weng [38].

The VIM system will support functional programming and the execution mechanism is based on data flow. In the world of functional programming all values are treated as mathematical values. This implies that the traditional view of data structures (arrays and records) as *modifiable* entities is no longer valid – the system must operate so that the user gets the view that a new structure is created from the old one whenever required. In a simplistic implementation, this would lead to a proliferation of copies of data structures, each differing from the others in only a small number of values. It is recognized that sharing of common elements among structures would reduce both the amount of copying and the storage space required to run the program. Various proposals have been made to implement data structures in data flow systems; none of them can be called definitive solutions. Applicative languages are also side-effect free languages and the language constructs provided in the functional language for VIM does not allow the creation of circular structures. Therefore, reference-counted memory managemenrt becomes an attractive alternative to traditional mark-and-sweep methods for garbage collection. This thesis proposes a representation for data structures that greatly reduces the amount of copying and describes a reference count mechanism for storage reclamation.

In most currently proposed functional language architectures, an implicit assumption is that the program and the data which the program operates upon are all located in the main memory[1]. VIM has a two-level physical hierarchy of storage consisting of a large, slow disk and a smaller, faster main store. Values in the main-memory can be accessed immediately while values which are resident in the disk must be read into the main memory first.

The problem of storage reclamation on systems with large address spaces is a prickly one; the strategy for garbage collection in VIM is based on reference counting. The architecture of VIM modelled in this thesis consists of a single processor, some main store and disk store. The principal source of parallelism in the single-processor version of VIM stems from the concurrency in the processing of instructions and disk activities.

1.1 The VIM Project

The goal of the VIM project is to develop a computing environment which supports functional programming and provides a large address space and automatic storage reclamation. A two-level physical storage has been chosen to reduce the cost of physical memory. The primary vehicle for programming on this system will be the VimVal language, a functional language that is an extension of the language VAL developed by Ackerman and Dennis [1, 26, 27]. The criteria that have guided the design of the new language are that it should have the following characteristics.

- It should be sufficiently expressive in that it provides language constructs to the programmer to express most application programs that he needs to, without having to appeal to some features of the underlying architecture that are not evidenced in the language.

- A program consists of one or more modules. Modules must be independently compilable. All the independently compiled modules of a program are linked prior to execution by a linker.

- The language must be strongly typed, *i.e.*, if the compiler and the linker certify the program to be legally typed then the program will not encounter any type errors at the time of execution of the program.

- The language must provide constructs to express computations on streams.

[1] The models proposed by Dennis and Weng have a two-level physical storage.

- Non-determinacy must be expressible in the language.
- Higher-order functions must be permitted.

Programs written in VIMVAL will be run on a data flow processor with hierarchical storage. The conceptual framework for the machine was described in [38]. As currently envisioned, the VIM system consists of a single processing element and a two-level physical storage consisting of main memory and disk.

1.2 Background and Previous Work

A number of projects have aimed at providing a coherent and structured programming environment within the framework of a multiprocessing system. Of principal interest from the perspective of this thesis are the Hydra/C.mmp system, the Cm* computer and the SYMBOL computer.

The Hydra/C.mmp was an experimental multiprocessor system [39]. Capabilities were adopted as a mechanism for providing a large and uniform address space and also to control accesses to shared data structures. However, the system fell short of providing a truly integrated interface between the capability architecture and the programming language. The task of processor management was left largely in the hands of the user. The user had to ensure the correct usage of shared data structures by the use of appropriate locking and synchronization primitives, with a resultant decrease in the programmability of the system [23]. However, in spite of these shortcomings the Hydra/C.mmp system represented a significant advance in programmability over the multiprocessor systems then existing.

The Cm* [34, 35] was also a capability based architecture consisting of a large number of processors and memory modules. An underlying goal of the Cm* project was to develop a system that would be scalable, i.e., the computing power of the system would grow in proportion to the number of processors in the system. However, this effort too left the issue of processor management as a user responsibility. Also, since the cost of a memory access was proportional to the distance of the memory cell from the processor, the task of organizing the program so that the number of non-local memory references would be minimal was left to the programmer [22].

The Mu project [18] at MIT was aimed at assessing the importance of programmability in multiprocessor organizations. Though the theoretical framework appears to provide a better working environment than in Cm* and C.mmp, the system was still unable to provide an elegant way of avoiding the need for explicit synchronizaton mechanisms for shared data that could be updated independently by the processors. It became clear from Halstead's work that the language supported by a multiprocessor is critical to the usability of the system. The difficulty of programming on a multiprocessor can be alleviated if the user can write programs without having to worry about task scheduling, process synchronization and hazards such as read-before-writes, such chores being taken care of by the underlying system automatically.

The SYMBOL computer system [8, 29] was a language based multiprocessor system that allowed the user to program without having to worry about low level considerations like mapping the tasks onto the processors. Each of the processors had a very specific task; however, the task division was so rigid that it ruled out the possibility of scaling the system. Also, the various processors did not aim at solving a single problem in parallel. There was a processor dedicated to compilation, one to memory management, one to I/O management, one that actually executed the compiled program, etc. The parallelism in this system resulted from the fact that memory management, input-output and actual processing could be done in parallel. There was no facility in the system whereby multiple processors could concurrently *execute* a compiled program.

The SYMBOL was not a true multiprocessor since it was unable to support parallel execution of a program exhibiting a lot of computational parallelism. However, many of the ideas it introduced were far ahead of the times. It was one of the first processors to specialize the memory architecture to support structure memory. The memory representations of data were specialized to reflect the type of data, allowing operations to be performed on such typed data more efficiently. Significant amount of specialized hardware was developed to allow structure operations to be executed fast — a revolutionary approach, considering the cost of hardware in that period.

One of the seminal contributions of the SYMBOL system is that it viewed that the design of the memory management system was an integral part of the multiprocessor system design. The memory management mechanism provided primitives which could support high-level memory abstractions such as stacks, queues, lists and strings. A specialized processor performed the memory management tasks, exemplifying the philosophy of static load distribution that so characterized the system.

In spite of its failings, the SYMBOL system, which predated the other projects discussed above by a number of years, presented a pointer to the direction in which the development of programming environments for multiprocessor systems ought to proceed – an architecture based on a high-level language that provided a very uniform, integrated environment for programming.

Among the various general purpose computing environments available on modern systems, the one on the Lisp machines deserve special mention. Lisp machines are language-based uniprocessors designed at Massachusetts Institute Technology [24, 28, 32, 33, 36]. They provide a uniform programming environment; there is no distinction between the command language used for interaction with the system and the principal programming language supported (Lisp), the hardware is tailored for processing Lisp primitives, high level data structures such as lists and arrays are regarded as data types even at the machine architecture level, and mechanisms for storage reclamation constitute an integral part of the system design.

Lisp machines provide a very large address space which can be effectively used to support a uniform addressing scheme for all objects created in the system. However, the necessity of explicitly "loading" a file containing an object residing in secondary storage before the object can be used detracts from the uniform addressability feature. Once the file is loaded, the object may be placed on the disk by the memory manager; references to this object are handled by the system so that its actual placement in the memory hierarchy is transparent to the user. Ideally, the user should never have to worry about whether the object is in the primary storage or in the secondary; given the name of the object, the system should automatically resolve the references to the object appropriately. In the VIM system, there is no concept of a file – all data structures are persistent in that they continue to exist across sessions, until there exist no references to the structure in the system, in which case they are discarded. This strategy obviates the necessity of "loading" files.

The storage reclamation scheme adopted by the Lisp machine is a variant of the mark and sweep strategy. The process of marking and sweeping the address space starts when the system runs out of storage. In the Lisp machine, the process of garbage collection is overlapped with the processing of other tasks in the system. This overlap is achieved by frequent switching of the task of reclamation with other activities. In conventional systems (von Neumann derived architectures), the switching of tasks involves the execution of substantial amounts of code to save the state, reducing the efficiency of the process of switching. The data flow model of program execution allows such switching to be performed with only a small overhead. In mark-and-sweep garbage collection, the time required to reclaim storage is proportional to the size of the address space over which the reclamation is to be performed. By constrast, the time required to reclaim the storage occupied by an object by reference-count based reclamation schemes is proportional to the size of the storage occupied by the reclaimed objects.

1.3 Outline of the Thesis

The design of the VIM architecture espouses the following philosophy. An architecture should be developed by successive refinement, starting from an abstract mathematical specification. The extensions and refinements at each of specification are designed to permit more efficient implementation of the machine. By proving that all the models are equivalent[2], one can largely eliminate the unexpected behaviours that one encounters when designing a system in an *ad hoc* basis. This type of top-down approach is especially important to the design of multiprocessor systems, since the possibility for errors of omission and commision is so much greater.

The design of the VIM system started with the design of the language VIMVAL which conformed to the aims outlined earlier. VIMVAL programs are compiled into programs in a base language, a preliminary version of which was proposed by Dennis and Stoy in 1982; a refinement of the base language is presented in this work.

In this thesis, first the operational semantics of an abstract model for VIM is described. The model, called L1, is the basis for specifying the behaviour of VIM, and is a set theoretic characterization of the abstract machine. The execution model is defined by a non-deterministic state-transition function. The set of instructions in this abstract model is an extension of that proposed by Dennis and Stoy. Chapter 2 gives a brief description of the VIMVAL language and then presents the formal model

[2] A naive notion of equivalence may be that the models produce the same results. A formal notion of equivalence for the various models of VIM will be defined in chapter 4

L1.

The operational model L2 is a refinement of L1 and is obtained by adding the notion of storage. In this model, structure values (such as arrays and records) which were modelled as elements of sets in L1 are viewed as stored values. L2 models a system with a hierarchically organized physical memory consisting of main store and disk. Storage consists of a large collection of equal sized chunks, each of which is an ordered set of words. Structure values are stored in trees of chunks, thus permitting sharing of information. L2 models a strategy for storage reclamation based on a reference count mechanism. The operational semantics of this model is presented in chapter 3.

In accordance with our proposal of designing by refinement, we must next demonstrate that the L2 satisfies the specifications of L1. This is shown by proving that the two machines are computationally equivalent. A formal definition of equivalence is developed in Chapter 4 and the proof of the equivalence of L1 and L2 is presented.

The base language for the machine, which is the target language of the compiler for VIMVAL, is described in Chapter 5. Essentially, the data flow graphs are such that when the computation of a program terminates, the reference counting mechanism would guarantee that if a structure becomes inaccessible in L1, the corresponding element in L2 would have reference count of zero and would thus be reclaimed.

The thesis concludes with a discussion of the relationship between L2 and its physical realization, and a brief list of related problems which are beyond the scope of this thesis and need further investigation.

Chapter Two

The Val Interpretive Machine

The goal of the VIM project is the design and development of a computer system that supports functional programming well. The architecture of the computer is based on data flow principles and the data flow model of program execution is well suited for interpreting functional languages.

The functional language suuported by VIM is VIMVAL, which has evolved from VAL. VIMVAL is a textual language and a brief description of it is given in the first section. VIMVAL programs are compiled into programs in the base language, which consists of a set of data flow graph schemata. Translation from VIMVAL to the base language is straightforward since each construct in VIMVAL corresponds to a graph schema in the base language.

Programs in the base language are executed by interpreting the data flow instructions which are the nodes in the data flow graph for the program. Section 2.3 gives an informal description of some of the distinctive mechanisms used in VIM. The operational semantics of the abstract model L1 is presented in section 2.4. The model is the specification of VIM and all implementations of VIM must meet the specifications.

2.1 The VIMVAL Language

The programming language for the VIM system is the VIMVAL, an applicative language which is a revision and an extension of the Val programming language. The extensions include the addition of stream-types, free variables, recursion and mutual recursion, and higher order functions. A type inference mechanism guarantees type safety even if most type declarations are absent. Type inference is also used to provide polymorphic functions.

The data types of VIMVAL fall into two classes — *simple* types and *structure* types. The simple types include the familiar types **integer**, **real**, **boolean**, **character** and **null**. The structure types include array-types, record-types, distinguished unions, stream-types, and functions.

Functions are first-class objects. They may be passed as arguments to and returned as results from functions, and they may be built into data structures. The body of a function definition is an expression. Evaluation of an expression yields a single value or a tuple of values. Forms of expressions include the conditional expression, the **tagcase** expression, and the function invocation. There is no form of expression for expressing iteration, use of recursion being preferred.

2.2 An Example Program in VIMVAL

A program in VIMVAL consists of one or more modules. Each module has a header specifying its interface, type declarations, function definitions and one expression which constitutes the body of the module. An example module is shown in Figures 1 and 2. Figure 1 illustrates how the user may define a new data type *List*, which represents a list of integers. The example module defines three simple operations on objects of type *List*. Figure 2 illustrates the use of streams in the language. The functions *car*, *cdr* and *cons* defined by the example programs have the same meaning as in Lisp. The function *ListToStream* creates a stream of integers when it is given a list of integers. *SumOfStream* sums up the elements of a stream of integers.

A module written in VIMVAL defines a function that may be invoked from within another module or by a user command to the system. A module may contain function definitions – these may be invoked only from within the module unless they are explicitly exported by incorporating them into data structures sent out as module results. The body of a module may use names that are not defined bound to values by definitions in the module. These *free* names must be bound to other modules before the module may be run.

Within a module, type declarations precede the function definitions and the body. Within a function, the type declarations must precede the expression that constitutes the body of the function. An array *A* of integers is declared as follows.

A : **array[integers] = array(1, 100)**

The elements of the array are initially undefined. **select**(*A*, *i*) returns the *Value* of the *i*th element of the array. **append**(*A*, *i*, *v*) creates a new array which is identical to the array *A* except that the value of the *i*th element of the new array is *v*.

```
module returns record[head, tail, tuple : function]
  type List = oneof [emptylist : nil;
                     atom : integer;
                     pair : record[first, second : List]]

  function car (L:List) returns List;
    tagcase L
      tag emptylist : error;
      tag atom : error;
      tag pair : L.first;
    endtag
  endfun

  function cdr (L:List) returns List;
    tagcase L
      tag emptylist : error;
      tag atom : error;
      tag pair : L.second;
    endtag
  endfun

  function cons (L1, L2 : List) returns List;
    make List[pair : record[first:L1, second : L2]]
  endfun

  record[head:car, tail:cdr, tuple:cons]
endmodule
```

Figure 1: An example program in VIMVAL.

The definition of a record-type is of the form

type *Pair* = **record**[*first*, *second* : *List*]

Records of type *Pair* have two fields named *left* and *right*. The operation

record[*first* : *v1*, *second* : *v2*]

constructs a record where *v1* and *v2* are of type *List*. Record fields are accessed by the select operation, for instance

L.first

yields the value of the *left* field of *L*, which must be of type *List* in which the tag is *pair*. Tagged unions are used where different choices of representation are appropriate for different cases of a value. For example, the type *List* is a tagged union.

```
function ListToStream (L:List, C:List) returns stream[integer]
   tagcase L
      tag empty : tagcase C
                     tag empty : stream[];
                     tag atom : affix(C, stream[]);
                     tag pair : ListToStream(car(L), cdr(L))
                  endtag
      tag atom : affix(L, ListToStream(car(L), cons(cdr(L), C));
      tag pair : ListToStream(car(L), cons(cdr(L), C))
   endtag
endfun;

function SumOfStream(S:stream[integer]) returns integer;
   if isempty(S) then 0
   else first(S) + SumOfStream(rest(S))
   endif
endfun;
```

Figure 2: Continuation of the example.

```
type List = oneof [emptylist : nil;
                   atom : integer;
                   pair : record [first, rest : List]]
```

where the subtypes are distinguished by the tags *emptylist*, *atom* and *pair*. **make**[*atom* : 0] creates a oneof in which the tag field is *atom* and the associated value is 0. A case expression is used to access values of a **oneof** type :

```
tagcase L
   tag emptylist : expr1;
   tag atom : expr2;
   tag pair : expr3
endtag
```

A stream is a sequence of values, all of the same type, that are passed in succession, one-at-a-time between functions. The operations defined on streams are [], **first**, **rest**, **affix** and **empty**. [] produces an empty stream. **first**(*S*) produces the first element of the stream *S*. The result of **rest**(*S*) is the stream left after removing the first element of *S*. **affix**(*v*, *S*) is the stream whose first element is *v* and whose remaining elements are the stream *S*. The result of **empty**(*S*) is true if *S* is an empty stream, false otherwise.

2.3 The VAL Interpretive Machine – VIM

The abstract architecture of VIM uses data driven program execution. A program in the base language consists of one or more functions, each represented by an acyclic, directed data flow graph. The nodes of the graph are the instructions and the arcs between the nodes specify the data dependencies among the instructions. Arcs connecting two nodes may be of two types – *value arcs* and *signal arcs*. Values are carried on *tokens* along the directed value arcs of the graph. A *function template* is an array of the instructions which belong to the data flow graph corresponding to a function definition in VIMVAL. The size of the array is equal to the number of instructions in the data flow graph and the indexing of the array starts from 1. Instructions are identified by their array indices within a function template.

In VIM, iteration is modelled as recursion, and the chosen method for implementing recursion avoids the use of cyclic graphs[3]. Instead, each function application uses a fresh copy of the graph represented by the function template, the copy being called an *activation template*. An instruction is *enabled* or ready for *firing* when a value is available on each input value arc, and a signal has been received on each signal arc. Note that it may happen that some instructions in a template[4] will receive values but will never fire because no signal will ever arrive. In chapter 5 we give rules of graph construction to ensure that this does not happen; otherwise, the storage reclamation scheme will be unable to reclaim all possible structures, leading to degraded memory utilization.

A salient characteristic of VIM is that no arc is ever reused – at most one value or signal will be sent from one instruction to another along a value or signal arc of an activation template, respectively. This is assured by the acyclic nature of the data flow graphs and by the property that each function application produces a new activation template. This is quite different from the data flow models used by the U-Interpreter [2] or the Static Data Flow machine [14, 12]. In the static data flow machine, the data flow graph does not change during program execution. The creation of function activations provides a very natural way of implementing recursion in VIM. VIM is similar to the static machine in that instructions have special fields for holding the operand values. This is quite unlike the mechanism

[3] Turner uses cyclic graphs to implement recursion in [37]

[4] We shall use "template" instead of "activation template" whenever there is no cause for confusion.

used in the U-interpreter where the value is stored in an associative store. Function application in VIM expands the execution graph due to the creation of activation templates; the graph contracts whenever a function terminates and the activation is discarded. In the U-interpreter function application results in the creation of a new context, which is a part of the tags on values.

Another feature of VIM which distinguishes it from other data flow models such as the U-interpreter or the Static data flow machine is the *heap*. VIM maintains a heap in which all objects except scalars that enter into computation are held. Scalar values are stored in the operand fields of the instructions, and passed around among the instructions on the tokens. The kinds of objects held by the heap include function templates, closures, early-completion queues (described below) and data structures (arrays, records, etc.). Each object on the heap has a unique identifier which permits its selection from among all objects in the heap. Conceptually, the heap is a multi-rooted, directed acyclic graph in which an arc signifies that the target object is a component of its superior.

A distinctive feature of VIM is the set of mechanisms designed to support aspects of the VIMVAL language; in particular, these include support for function application and tail recursion and computation on streams. These mechanisms are described informally below; a formal description of the mechanisms will be presented in the next section.

2.3.1 Function Application

Function applications are made by the APPLY instruction, which requires two operands — a function closure for the function to be applied, and a data structure containing argument values. The first element of the closure is the uid of the function template which is to be applied; the rest of the closure contains information defining the binding of any free variables of the function. The APPLY instruction creates an *activation* of the function by copying the function template. It then sends the closure, the argument structure and the return link to the first operand of the first three instructions in the activation template, respectively. The return link consists of the uid of the calling activation and the uid of the destination list of APPLY.

Instructions of the activation are then executed according to the data flow firing rule until the RETURN instruction is enabled. The RETURN instruction uses the return link to send the result of the function invocation to the recipients. Due to the presence of early-completion structures the RETURN instruction may not be the last instruction to execute in the activation. A separate RELEASE instruction releases the storage occupied by the activation template.

The following notation will be used for drawing data flow graphs. The nodes of a function template are instructions drawn as rectangular boxes. The value arcs connect from bottoms to tops of instruction boxes and convey data values. The signal arcs convey signals that perform control functions such as the release of function templates. The signal arcs connect from right sides to left sides of instruction boxes. Numerals at the left corner of instruction boxes denote the index of the instruction in the activation. A Greek letter next to an instruction box corresponds to the address of the instruction, consisting of the uid of the activation template and the index of the instruction. An open box with two or more values or signals is the merge operator. The graphs are arranged such that exactly one *Value* or signal will arrive at a merge box. This is merely a notational convenience; in VIM, the signal count and operand counts are set such that the merge occurs naturally.

Figure 3(a) shows a data flow graph which causes a function activation. λ is the address of the destination instruction of APPLY, which is sent to the third instruction of the activation created by APPLY. Figure 3(b) shows a typical function template. The RETURN instruction receives the destination list consisting of the address λ; when it receives the result computed by the function body, it sends the result to the instructions whose addresses are listed in the return link and sends a signal to a RELEASE instruction.

In many cases the value returned by a function *f* is computed directly by a tail-recursive application of *f*, as shown in Figure 4. In this situation the result to be returned by the caller is exactly that returned from the callee, and the reactivation of the caller is unnecessary. The TAILAPPLY instruction in VIM implements this. It also causes a function activation but is different from APPLY: it has an extra operand, a return link which it passes to the callee instead of generating a new one; also, it sends signals to the instructions whose indices are in the destination list of the TAILAPPLY instruction.

Figure 4(a) shows the TAILAPPLY instruction and illustrates the operands that it needs. Figure 4(b) shows a typical template corresponding to a tail-recursive function. The SWITCH instruction takes two operands: if its second operand, wheih must be boolean, is true then the first operand is sent to all

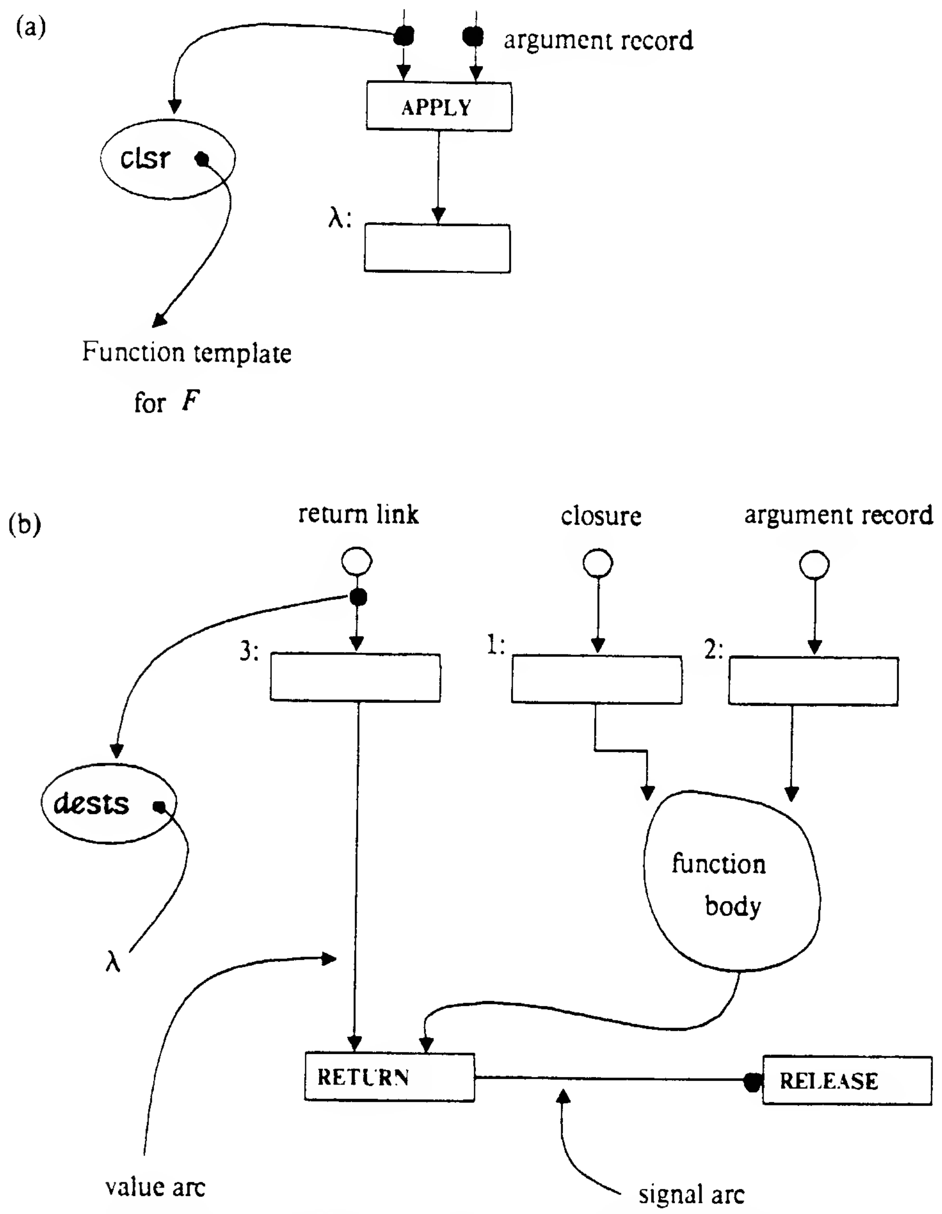


Figure 3: (a) shows the data flow graph for function invocation (b) Data flow graph of a typical function template. λ is the address of the destination of APPLY: it is a pair consisting of the uid of the calling activation and the index of the instruction in the template.

the destinations of SWITCH whose addresses are marked **true** in its destination list and if it is false then the first operand is sent to the destinations marked **false**. The function body determines if no further applications are to occur, in which RETURN is activated, otherwise the TAIL APPLY instruction is enabled.

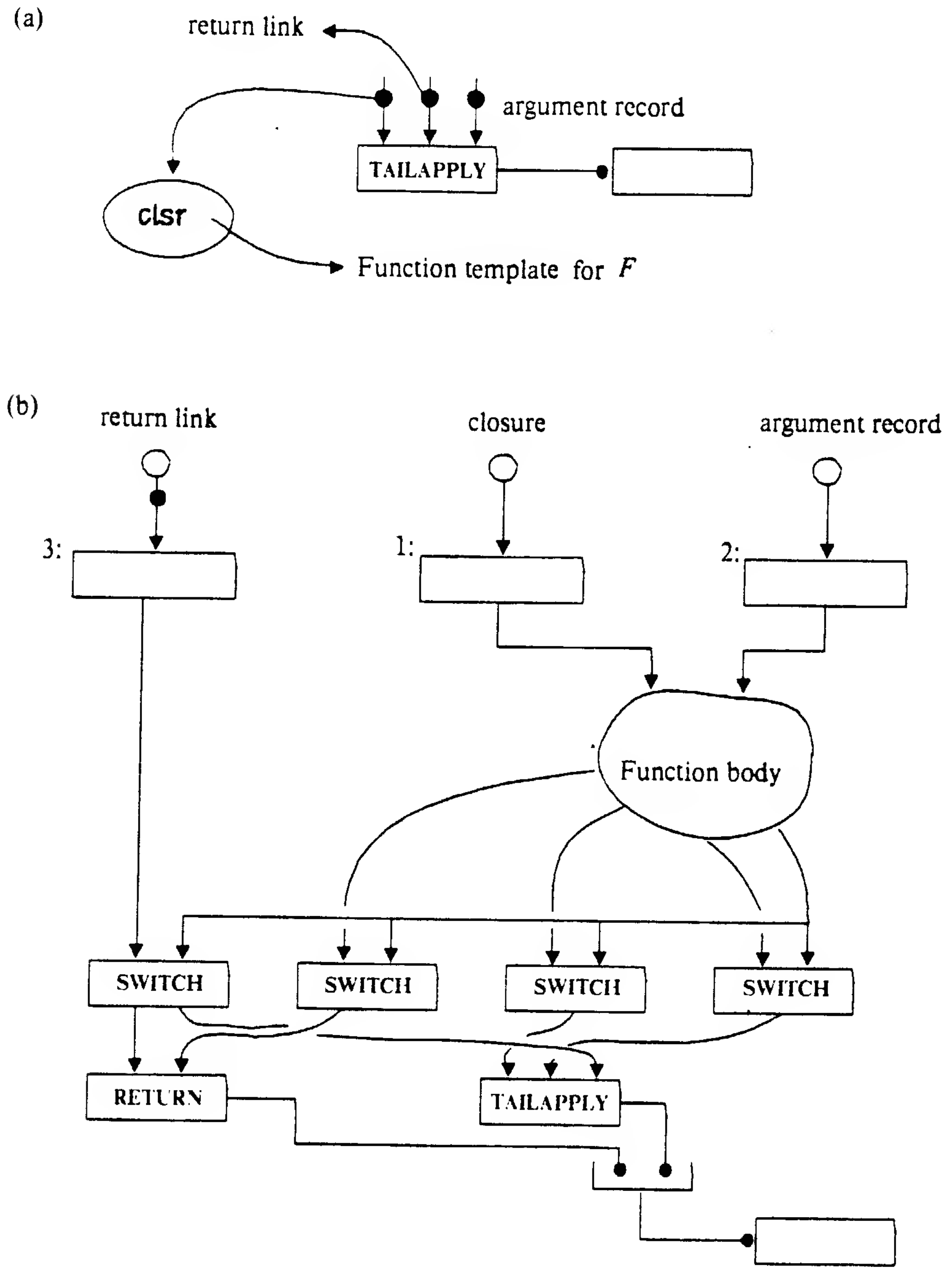


Figure 4: (a) The TAILAPPLY instruction. (b) Typical data flow graph for the body of a tail-recursive function.

2.3.2 Early-Completion Queues

In computations involving data structures, concurrency is increased if a data structure can be made available for access before all the component values have been computed. If instructions are required to receive all their operands before their application, as is usual for the execution of data flow programs, this concurrency of creating and accessing a data structure is not possible.

In VIM there is a special facility called early-completion queue (abbreviated EC-queue) to permit structures to be created before the values of all the components are available. Arrays will be used to describe the early-completion mechanism informally (figure (5). The behaviour of structures containing EC-queues is specified by the state-transition rules of the MKINARRAYEC, SELECT, APPEND and SET instructions.

An EC-queue is a collection of addresses of instructions. MKINARRAYEC creates an array in which all the elements are EC-queues, all initially empty. This shell of the structure is passed onto consumers of the data structure, and also to producers which replace the EC-queues by values using the SET instruction. If a SELECT tries to access an element which is an EC-queue, its address is added to the EC-queue and the instruction is removed from the set of enabled instructions. Eventually, a SET instruction replaces the EC-queue by a *Value* and adds the addresses of the instructions in the EC-queue to the set of enabled instructions. When these instructions are attempted for execution again, they would read the value, as desired. Structures with EC-queues provide a powerful mechanism for synchronisation, and is an effective solution to the read-before-write problem [4].

Figure 5 illustrates the early-completion mechanism. Figure 5(a) shows a data flow graph which creates an array of one element which is an empty EC-queue. The array is sent to two consumers whose addresses are α and β, and to a set instruction γ. Figure 5(b) shows how the contents of the array changes when the instructions α, β and γ fire in sequence. If γ fires first, then α and β can access the value in the usual manner; the erstwhile presence of the EC-queue does not affect subsequent accesses after it is replaced.

The early-completion mechanism makes it possible to allow function applications to begin execution before the values of all their arguments have been computed. This is done by packaging the arguments into a record of EC-elements. Similarly, the result values, if there are more than one, may be returned as a structure of EC-elements so each may be available to the caller without waiting for all the results to be evaluated.

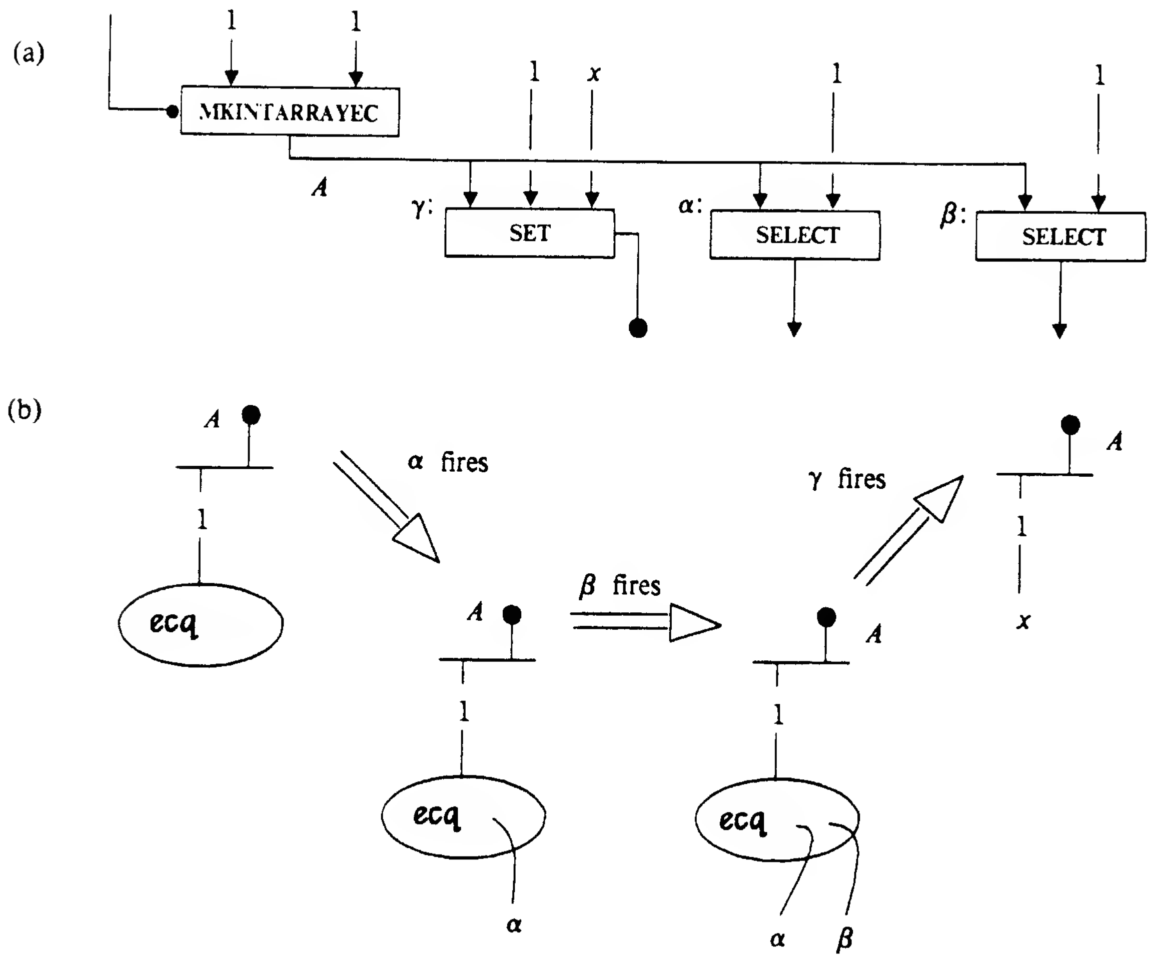


Figure 5: (a) Data flow graph showing producer-consumer relationship for structure containing EC-queue. (b) The contents of the array under the firing of α, β and γ in sequence. First, the firing of α causes α to be added to the EC-queue, which was empty. Next β attempts to access the element and also gets added to the EC-queue. Eventually when x is computed, the SET fires. It replaces the EC-queue with x and adds α and β to the set of enabled instructions.

The semantics of arrays with ecqueues is very similar to the semantics of *I-structures*, which were proposed by Arvind and Thomas [5]. An I-structure is a linear contiguous data structure: an element of an I-structure can be written into at most once. Reads occurring before an element has been written into are deferred until the arrival of the value. The *futures* construct of Halstead [19] is also of similar flavour.

2.3.3 Suspensions and Streams

Stream structures are an attractive language feature since they permit the producer and consumers of the stream to operate concurrently. VIM provides a special mechanism for efficient implementation of streams. A stream is represented in VIM as a chain of records of two elements, each of which is an EC-queue; the first element contains an element of the stream and the second element is a pointer to the rest of the stream. In a completely data driven evaluation of a stream, the producer would proceed at its own pace and generate the values to construct the stream. The consumer process accesses the elements of the stream at its own rate, waiting whenever it encounters an EC-queue until the value is supplied.

The problem with this scheme is that it allows the producer to get arbitrarily far ahead of the consumer process. If the consumer needs only a part of stream then substantial computation performed by the producer may wasted. In particular, if streams are evaluated in a data driven manner, then infinite streams cannot be supported on VIM. So streams are produced in a data driven manner, allowing the user to write programs in VIMVAL which deal with potentially infinite streams.

VIM uses *suspensions* to implement demand-driven evaluation of streams. Suspension mechanisms have been used to implement infinite data structures by Henderson [20], Friedman and Wise [16], etc. In VIM, a suspension contains the address of an instruction, consisting of the uid of the activation template of the instruction and its index in the template. When a SELECT instruction tries to access an element which is a suspension, the suspension is replaced by an EC-queue containing the address of the SELECT and a signal is sent to the instruction whose address is found in the suspension. The signalled instruction eventually causes the EC-queue to be eventually replaced by a value and the SELECT instruction gets the value it was trying to access.

Figure 6(a) shows the creation of an array whose only element is an EC-queue. If SETSUSP fires before SELECT, it finds an empty EC-queue and replaces it by a suspension. When SELECT executes, the suspension is replaced by an EC-queue containing the address β of SELECT and a signal is sent to the instruction indicated in the suspension. If SELECT fires first, it is enqueued in the EC-queue; SETSUSP executes, finds a non-empty EC-queue, and simply sends a signal. The graph is arranged such that the arrival of the signal initiates a computation that ultimately enables a SET instruction. The SET instruction replaces the EC-queue in this array by a value and services the EC-queue by adding the set of instructions whose addresses are in the EC-queue to the set of enabled instructions.

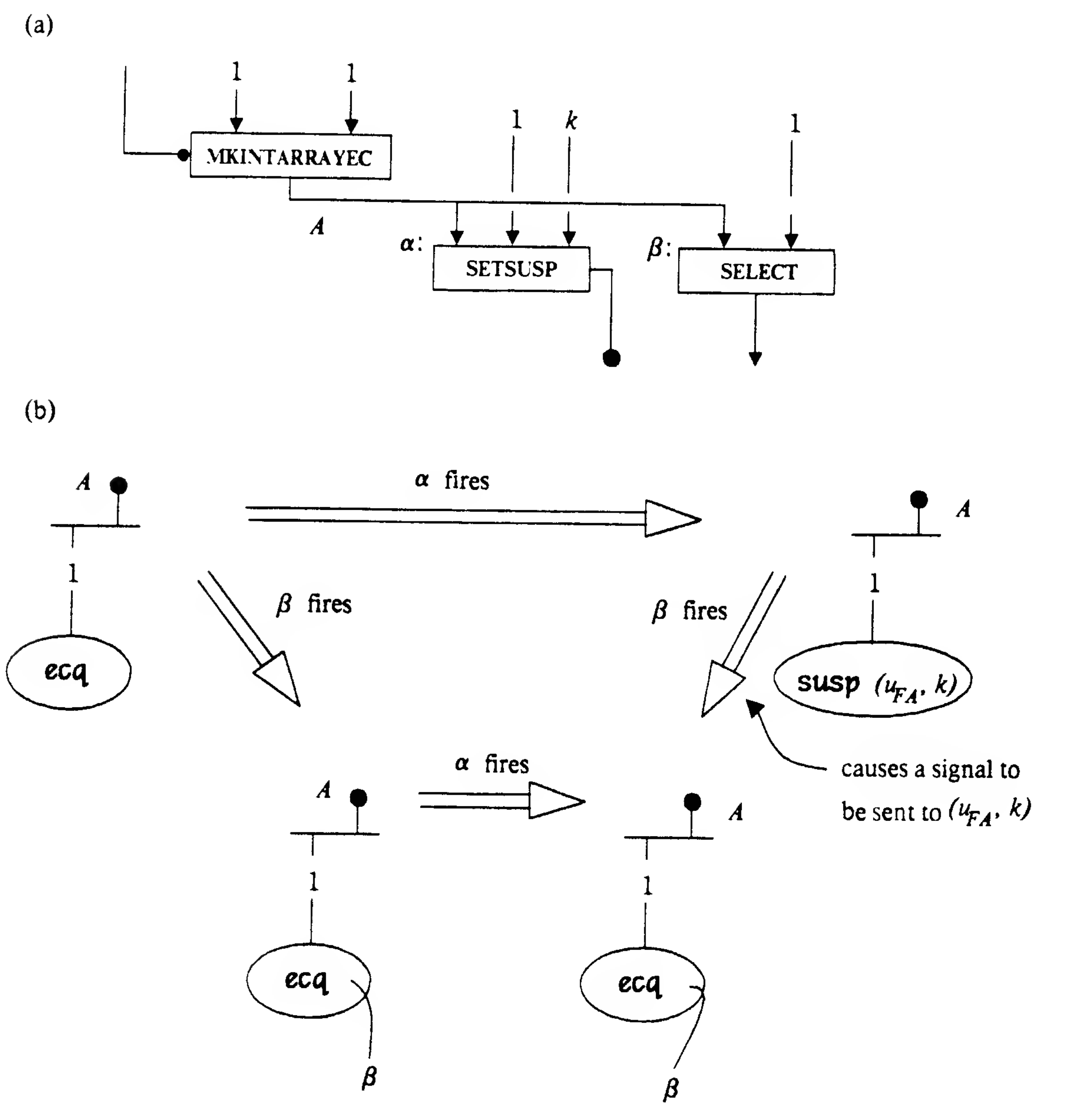


Figure 6: (a) Creation of an array with a suspension element. (b) Effect of firing of SELECT and SETSUSP instructions in different orders.

The use of suspensions for generating the elements of a stream is shown in Figure 7. The records which constitute the stream have two fields which are named *head* and *tail*. The stream consists of successive integers. Figure 7(a) shows an initial stream whose first element is 1 and the tail component is a suspension. When a consumer α tries to access the tail of the stream, the suspension is replaced by an EC-queue containing the address of the consumer (7(b)). A signal is sent to the suspended instruction which causes the EC-queue to be replaced by new record whose *head* component contains the next element of the stream and the tail component is a suspension (7(c)). The consumer which tries to access the rest of this stream in turn replaces the suspension by an EC-queue. If there are no consumers which have pointers to the beginning of the stream then the element at the front may be abandoned (7(d)). Suspensions can also be used to advantage for evaluating the elements of arrays in a demand driven manner. The main benefit in doing this would be that array elements which are never read need not be computed, thus reducing the amount of computation performed.

The rest of the chapter gives a mathematical specification of VIM. The operational semantics of the instructions of VIM are presented. The specification will be called L1 in this thesis. L1 will serve as the basis for the development of an operational model for VIM in which storage is modelled; that model will called L2.

2.4 Operational Model for VIM - L1

The VIM interpreter has two components : a function *Interp* and *State*. *Interp* takes two arguments — a *State* and an enabled instruction (defined later) and produces a new *State* of the machine. The following notation will be used in the thesis. Sets are denoted by bold font, elements of sets (which may themselves be sets) are denoted by italicised letters and names are indicated in a distinctive font. Thus, **This** is a set, *This* ∈ **This** and This is a name.

The actions of the interpreter are described by state transition rules. A programming language-ish description is used to specify the rules for mapping a set to another set. Mathematical notations such as set unions and differences are used wherever convenient. A rule F is expressed as follows:

```
Define $F(A, (i, j), v) \equiv$
    ... body of the rule
```

The definitions may be recursive.

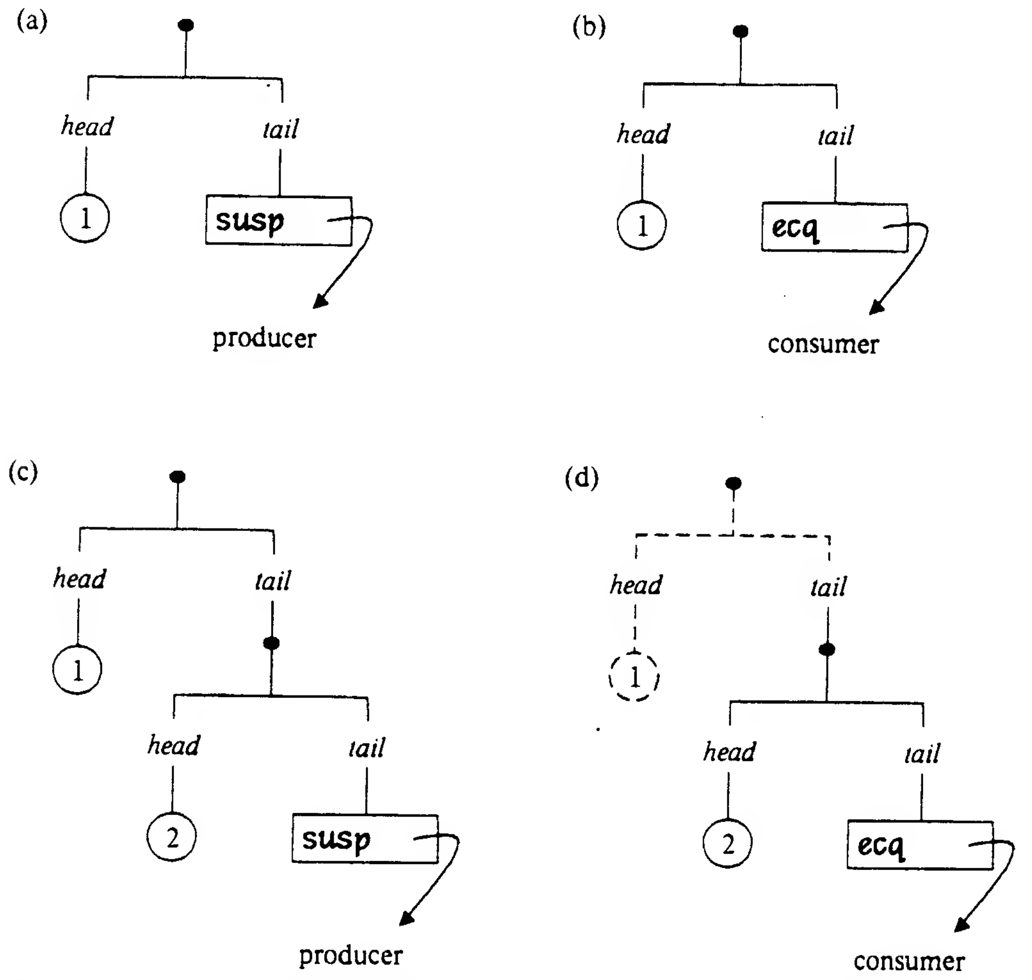


Figure 7: Demand-driven generation of stream elements. (a) Stream element: the producer is awaiting a demand. (b) The consumer demands the next stream element. (c) The producer generates one stream element and suspends itself. (d) The consumer abandons the previous element and demands another.

VIM = <*Interp*, *State*> where
- *Interp* : State × EIS → State
- State = Act × H × EIS,
- Act = U → Function
- H = U → ST
- U = the set of all unique identifiers,
- EIS = the set of all enabled instructions, described later.

Act is the set of all activations: an activation is created by the invocation of a function. The heap H contains all structure values and function templates, early-completion queues (discussed later),

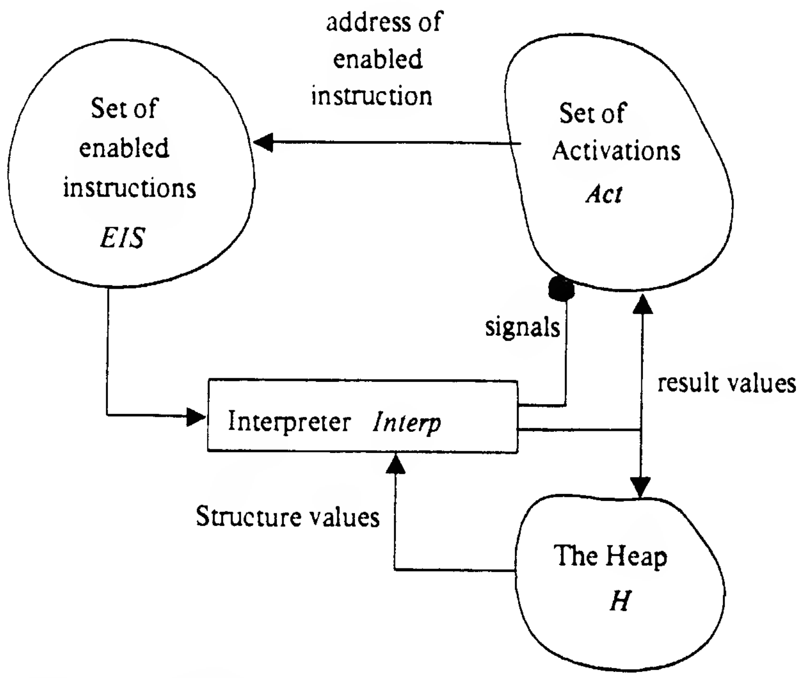


Figure 8: The abstract VIM architecture

function closures and the instructions. Each element on the heap has a unique identifier (uid). Only scalar values and uids are sent on tokens from one data flow actor to another; data structures always reside on the heap.

Scalar values are tagged.

Scalars = **Integers** ∪ **Reals** ∪ **Booleans** ∪ **Character** ∪ **Null**
Integers = {**int**} × ({*undef*} ∪ the set of all integers)
Reals = {**real**} × ({*undef*} ∪ the set of all reals)
Booleans = {**bool**} × ({*true, false, undef*})
Character = {**char**} × ({*undef*} ∪ the set of characters in the machine.)
Null = {**null**} × {*nil, undef*}

The set **ST** describes the elements which reside on the heap. Elements of different types are distinguished by their tags.

$$\begin{aligned} ST = & (\{\mathbf{arr}\}\times(\text{Array}\cup\{\mathit{undef}\})) \cup (\{\mathbf{fn}\}\times\text{Function}) \\ & \cup (\{\mathbf{ecq}\times\text{ECQ}) \cup (\{\mathbf{inst}\}\times\text{Instruction}) \cup \{\mathbf{dests}\}\times \text{Dests} \\ & \cup (\{\mathbf{clsr}\}\times\text{Clsr}) \end{aligned}$$

Array = [Integers → (U ∪ Scalars ∪ SUSP], Integers being the set of integers.

Function = [N → Instruction], N being the set of natural numbers.

An early-completion element (EC-element)is a tuple $(u,\ i)$ where u is the uid of a function activation and i is the index of an instruction in the activation. An early-completion queue is a collection of such EC-elements.

$$\text{ECE} = \text{U} \times \text{N}.$$

Thus $(u,\ k) \in$ ECE where u corresponds to the uid of a function activation and N is the index of the instruction in the activation template.

The EC-queue is a collection of elements of ECE. All EC-queues are members of the set **ECQ** which is defined below. The notation $\mathcal{P}$(N) denotes the powerset of the set **N**.

$$\text{ECQ} = \mathcal{P}(\text{ECE})$$

A suspension is a member of the set SUSP specified by :

$$\text{SUSP} = \{\mathbf{susp}\} \times (\text{U} \times \text{N})$$

An instruction is a seven-tuple :

$$\text{Instruction} = \text{OPS} \times (\text{U} \cup \text{Scalars})^3 \times \text{N} \times \text{N} \times \text{U}$$

OPS is the set of opcodes, the next three elements of the tuple refer to the operands, the fifth and sixth elements of the tuple are the operand count and the signal count and the last element is the unique identifier of the list of destinations. Fach destination of an instruction is the index of the instruction to which the result is to be sent. The result may be a value or a signal. For $I \in$ I, the elements of the tuple will be denoted by the '.' notation. Thus, *I*.**opcode** is the first element of the tuple, *I*.**op1**, *I*.**op2** and *I*.**op3** refer to the second through fourth elements of the tuple, I.**opcnt**, *I*.**sigcnt** and *I*.**destlist** denote the fifth through seventh elements of the tuple.

OPS = {IADD, ISUB , MKINTARRAY, MKINTARRAYEC, SELECT, APPEND, SET, SETSUSP, SWITCH, , APPLY, TAILAPPLY, RETURN, RELEASE}

A destination of an instruction consists of the the address of the instruction to which the result is to be sent, and the operand which is to receive the result value. **op1**, **op2** and **op3** denote the first, second and third operand fields in an instruction, respectively. If the result is a signal then no operand number is required. The destination also specifies if the result is to be sent unconditionally or conditionally. For all instructions except for SWITCH, the results of instructions (both values and signals) are sent to the destinations unconditionally.

$\text{Dests} = \mathcal{P}(\text{D})$

$\text{D} = \{\textbf{unconditional}, \textbf{true}, \textbf{false}\} \times \text{N} \times \{\textbf{op1}, \textbf{op2}, \textbf{op3}, \textbf{signal}\}$

Clsr is the set of closure records. The operator (APPLY, TAILAPPLY, STREAM-APPLY) which cause a function activation take a closure as the first argument. The first component of the closure is the uid of a function template; that uid corresponds to the function which is to be called.

$\text{Clsr} = [(\{\textbf{FunctionToApply}\} \cup \text{M}) \rightarrow (\text{U} \cup \textbf{Scalars}]$
where $C(\textbf{FunctionToApply}) \in \text{U}$ such that $H(u) \in (\{\textbf{fn}\} \times \textbf{Function})$.

An enabled instruction is an element of $\text{EI} \subseteq \text{U} \times \text{N}$. An instruction I becomes enabled when $I.\textbf{opcnt}$ and $I.\textbf{sigcnt}$ both become zero. The set of enabled instructions describes the collection of instructions which are ready to executed because they have received an operand on each of the operand arcs and a signal on each of the signal arcs. The set of enabled instructions is:

$\text{EIS} = \mathcal{P}(\text{EI})$

The function *Choice* selects an element from a set of enabled instruction. The instruction is then interpreted by the function *Interp*. Thus *Choice* is our scheduler:

$Choice : \text{EIS} \rightarrow \text{EI}$

Functions *AddToHeap* and *DeleteFromHeap* add and delete elements from the heap. *AddToAct* and *DeleteFromAct* which

$AddToHeap : \text{H} \times \text{U} \times \text{ST} \rightarrow \text{H}$

$AddToHeap(H, u, v)$ produces a new function H' such that :

$(\forall\ u' \neq u\,[H'(u') = H(u')])$ and $H'(u) = V$

$DeleteFromHeap : \text{H} \times \text{U} \times \text{ST} \rightarrow \text{H}$ $DeleteFromHeap(H, u, V)$ produces a new heap H' such that the domain of H' is the domain of H without the element u.

H' such that $(\forall\ u' \neq u\,[H'(u') = H(u')])$

AddToHeap(H, u, v) creates a new heap which contains, in addition to the associations between uids and objects in H, a new association between u and v. *DeleteFromHeap*(H, u, v) creates a new heap which does not contain the uid u in its domain.

Similarly, *AddToAct* and *DeleteFromAct* create new activation sets by adding an activation to and deleting an activation from the current set of activations.

AddToAct : **Act** $\times$ **U** $\times$ **Function** $\rightarrow$ **Act**
DeleteFromAct : **Act** $\times$ **U** $\times$ **Function** $\rightarrow$ **Act**

Function *SendResult* is used to dispatch the result of an instruction I to a destination instruction. *SendResult* models the following actions : the result is stored in the appropriate operand field of the destination instruction, and the operand count and signal count fields are decremented accordingly. Since we are dealing with a mathematical representation of instructions and function activations, this updating is modelled by producing new values that reflect the changes. Thus, I' is the destination instruction after it receives the result, FA' is the new new value of FA with the updated I' and Act' is the same as Act except that the ith instruction of activation FA has received some operand (or signal).

SendResult : **Act** $\times$ **EIS** $\times$ **U** $\times$ **D** $\times$ [[{value} $\times$ (**U** $\cup$ **Scalars**)] $\cup$ {signal}]
$\rightarrow$ **Act** $\times$ **EIS**

```
define SendResult(Act, EIS, $u_{FA}$, (dc, i, opnum), result) ≡

let
  $FA = Act(u_{FA})$,        % FA is an activation template.
  $I = FA(i)$                % I is the ith instruction in the activation template.
in
    $AddToAct(DeleteFromAct(Act, u_{FA}, FA), u_{FA}, FA')$,
    if $(I'.\mathrm{opcnt} = 0) \wedge (I'.\mathrm{sigcnt} = 0)$ then $EIS \cup \{(u_{FA}, i)\}$
    else $EIS$
    endif
endlet
where
    $FA'(j) = FA(j)$,      $j \neq i$.
            $= I'$,        $j = i$.
```

```
and  $I' \in$ Instruction,
    $I'$.opcode = $I$.opcode,
    $I'$.op1 = if $opnum \neq$ op1 then $I$.op1 else $t$ where $result$ = (value, $t$)
    $I'$.op2 = if $opnum \neq$ op2 then $I$.op2 else $t$ where $result$ = (value, $t$)
    $I'$.op3 = if $opnum \neq$ op3 then $I$.op3 else $t$ where $result$ = (value, $t$)
    $I'$.opcnt = if $opnum \in$ {op1, op2, op3} then $I$.opcnt else $I$.opcnt - 1
    $I'$.sigcnt = if $opnum$ = signal then $I$.sigcnt-1 else $I$.sigcnt
    $I'$.destlist = $I$.destlist
```

The function *SendToDestinations* sends the result of an instruction to all the destinations of the instruction. It is a simple tail-recursive function which uses *SendResult* repeatedly to send the value or signal to the destination.

$$SendToDestinations : \text{Act} \times \text{EIS} \times \text{U} \times \mathcal{P}(\text{D} \times ([\{\text{value}\} \times (\text{U} \cup \text{Scalars})] \cup \{\text{signal}\}) \rightarrow \text{Act} \times EIS$$

```
define SendToDestinations (Act, EIS, u, DestValue) ≡
    if DestValue = {} then Act, EIS
    else
      let ((dc, d, opnum), V) = e where $e \in$ DestValue
      in
        if V = signal then
          SendToDestinations
            (SendResult(Act, EIS, u, (dc, d, signal), signal), DestValue − {e})
        else
          SendToDestinations
            (SendResult(Act, EIS, u, (dc, d, opnum), v), DestValue − {e})
        endif
      endlet
    endif
```

Execution of a program on VIM is initiated by the invocation of a function in the base language. The execution terminates when there are no more enabled insrtuctions. The execution loop may be summed as :

```
define MainLoop (State) ≡
    let (Act, H, EIS) = State
    in
     if EIS = {} then halt
     else MainLoop(Interp(State, Choice(EIS)))
     endif
    endlet
```

The interpreter is defined by the function *Interp* and *Choice* is a function which selects an element from the set of enabled instructions. This instruction is interpreted by *Interp* in the context of the current state of the machine; the result of the execution of the instruction is a new state. *Choice* is the scheduler in VIM since it makes the choice of the instruction which is to be executed. *Choice*(*EIS*) where $EIS \in$ EIS is the address of an enabled instruction.

$$Interp : \text{State} \times Choice(\text{EIS}) \rightarrow \textbf{State}$$
$$\text{where } \textbf{State} = \text{Act} \times \textbf{H} \times \textbf{EIS}.$$

It is pertinent to point out that VIM is a non-deterministic state transition system; any one of the enabled instructions could be selected for execution and the final result of the computation is independent of the order of execution of the enabled instructions.

The following notation is used to denote that the new state (Act', H', EIS') is produced when the instruction *e* is interpreted by *Interp* in the state (Act, H, EIS).

$$(Act, H, EIS) \vdash (Act', H', EIS') \text{ on } e.$$

Now we can define the interpreter by specifying the state transitions for each of the opcodes. The state transitions for some of the more interesting instructions will be presented below; these serve as the model for specifying the transition rules for the rest of the instruction set. The body of *Interp* is a big conditional statement; the branches of the conditional are based on the opcodes of the instruction being executed. Some general comments are in order here. The result of an instruction is sent to its indicated destinations unconditionally, unless it is a SWITCH instruction.

```
define $Interp(State, (u_{FA}, k_{FA})) \equiv$
    let
      $FA = Act(u_{FA})$,        % the function activation
      $I = FA(k_{FA})$,        % the instruction
      $\{(dc_1, d_1, opnum_1), (dc_2, d_2, opnum_2), \ldots, (dc_n, d_n, opnum_n)\} = I$.destlist
                % the destinations of the instruction $I$.
    in
      if $I$.opcode = SET then ...
      elseif $I$.opcode = APPLY then ...
          .
          .
          .
      endif
    endlet
```

The specification of the state transition rules for some of the interesting and representative instructions of VIM will be the subject of the rest of this chapter. Each conditional statement of the form "if *I*.**opcode** = ... then " is an arm of the big conditional statement in *Interp* above. Thus, the names *Act*, *H*, *EIS*, u_{FA}, k_{FA}, (dc_1, d_1, $opnum_1$), ..., (dc_n, d_n, $opnum_n$) will have the same bindings as indicated in the body of *Interp* shown above.

Let us begin with the simple instruction IADD which adds two integers. The operands are read from the operand fields of the instruction and the result of the addition is *sum*. This value is sent to the listed destinations. Observe that the heap remains unchanged.

```
if I.opcode = IADD then
    let
     (int, m) = I.op1,
     (int, n) = I.op2,
     sum = m+n,
     Act', EIS' =
        SendToDestinations(Act, EIS, $u_{FA}$, {((unconditional, $d_1$, $opnum_1$), $\alpha_1$), ...,
                         ((unconditional, $d_n$, $opnum_n$), $\alpha_n$)})
    in
     Act',
     H,
     EIS - {($u_{FA}$, $k_{FA}$)}
    endlet
endif
```

where $\alpha_i \in$ {(**value**, *sum*), **signal**}

The actions of the interpreter for instructions such as ISUB, IMUL, IDIV, IGT, ILT, etc. are very similar and will not be described.

The instructions which operate on structures produce a new heap. The operations on one type of structures – arrays – are described here; the actions of *Interp* for instructions which operate on **record** and **oneof** types are very similar and are not presented. The array instructions of interest which are discussed below are : MKARRAYINT, MKARRAYINTEC, SELECT, APPEND, SET and SETSUSP.

MKARRAYINT takes two integer operands (*m* and *n*) and adds an array with bounds (*m*, *n*) to the heap. All the elements of the **array** are undefined. The uid of the new array is sent as the result to the destinations, along with signals if necessary.

```
if $I$.opcode = MKINTARRAY then
    let
        (int, $p$) = $I$.op1,
        (int, $q$) = $I$.op2,        % $p$ and $q$ are integers, $p < q$.
        $u_v$ = a new uid in U,
        $Act', EIS'$ =
            $SendToDestinations(Act, EIS, u_{FA}$, {((unconditional, $d_1$, $opnum_1$), $\alpha_1$), ...,
                        ((unconditional, $d_n$, $opnum_n$), $\alpha_n$)})
    in
        $Act'$,
        $AddToHeap(H, u_v$, (arr, ([$p$, $q$] $\rightarrow$ $undef$))),
        $EIS' - \{(u_{FA}, k_{FA})\}$
    endlet
endif
        where $\alpha_i$ is either (value, $u_v$) or signal.
```

The instruction MKINTARRAYEC works quite similarly except that the elements of the array are all early-completion queues, all empty.

The APPEND instruction takes three operands – an array A, an index i and a value x. It creates a new array A' which is identical to A except that the i th element of A' has value x. It is important to be very careful while performing APPEND operations on arrays with EC-elements. If some elements in A are EC-elements then the corresponding elements of A' would also be EC-elements. When a SET instructions replaces an EC-element in A by a value, this value must be forwarded to the corresponding elements in structures which were created by APPEND on A. There may be a cascade of value-forwarding precipitated by this since the values may also have to be forwarded to arrays created by APPENDs on structures derived from A. Since suspensions are potential sites for EC-queues, APPEND operations on arrays containing suspensions introduces a similar need for *Value* forwarding. This thesis adopts a simple alternative to the value-forwarding discipline outlined above. An APPEND instruction is executed provided that there are no EC-elements or suspensions in the array on which the operation is to be performed. If there is any EC-element in the array then there is no change in the state of the machine: the APPEND instruction remains in the set of enabled instructions and will be selected for execution at some future time.

```
if $I$.opcode = APPEND then
    let
     $u = I.\mathrm{op1}$,
     $(\mathbf{arr}, A) = H(u)$,
     $(\mathbf{int}, i) = i.\mathrm{op2}$,
     $x = I.\mathrm{op3}$,
     $u'$ = new uid from U
     $Act', EIS'$ =
        $SendToDestinations(Act, EIS, u_{FA}, \{((\mathbf{unconditional}, d_1, opnum_1), \alpha_1), \ldots,$
                              $((\mathbf{unconditional}, d_n, opnum_n), \alpha_n)\})$
   in
     if $|\{i : A(i) \in U \times \{(\{\mathbf{ecq}\} \times ECQ) \cup (\{\mathbf{susp}\} \times SUSP)\}| = 0$ then
      $Act'$,
      $AddToHeap(H, u', A')$,
      $EIS' - \{(u_{FA}, k_{FA})\}$
     else
      $Act$,
      $H$,
      $EIS$
     endif
   endlet
       where $A'(j) = A(j)$          $j \neq i$
                    $= x$             $j = i$.

       and    where $\alpha_i$ is either $(\mathbf{value}, u')$ or $\mathbf{signal}$.
```

SELECT requires two operands – the uid of an array A and an integer i, and in the simplest case (the element being accessed is neither an early-completion structure nor a suspension) returns the value associated with the element of A that has index i. The behaviour of the interpreter is more complex when such is not the case. The state transition is specified below, and the discussion on early-completion elements and suspensions follows. The special value *est* is used for indicating the end of a stream.

```
if $I$.opcode = SELECT then
    let
     $u = I.\mathrm{op1}$),
     $(\mathrm{arr}, A) = H(u)$,          % $(u_r\ (\mathrm{arr}, A)) \in H$
     $(\mathrm{int}, i) = I.\mathrm{op2}$,           % $i$ must be an integer
     $t = A(i)$)

    in
     if $t = (u_v, (\mathrm{ecq}, Q)$ then
        $Act$,
        $AddToHeap(DeleteFromHeap(H, u_v, (\mathrm{ecq}, Q)), u_v, (\mathrm{ecq}, Q \cup \{(u_{FA}, k_{FA})\}))$)
        $EIS - \{(u_{FA}, k_{FA})\}$
     elsif $t = (u_v, (\mathrm{susp}, (u', k')))$ then
        let $Act', EIS' = SendResult(Act, EIS, u', (\mathrm{unconditional}, k', \mathrm{signal}), \mathrm{signal})$)
        in
         $Act'$,
         $AddToHeap(DeleteFromHeap(H, u_v, (\mathrm{susp}, u', k')), u_v, (\mathrm{ecq}, u_{FA}, k_{FA}))$,
         $EIS' - \{(u_{FA}, k_{FA})\}$
        endlet
    else
        let $x$ =
          $Act', EIS'$ =
            $SendToDestinations(Act, EIS, u_{FA}, \{((\mathrm{unconditional}, d_1, opnum_1), \alpha_1), \ldots,$
                        $((\mathrm{unconditional}, d_n, opnum_n), \alpha_n)\})$
        in
         $Act'$,
         $H$,
         $EIS' - \{(u_{FA}, k_{FA})\}$
        endlet
endif
```

where α_i = either (value, x) or signal.

```
if $I$.opcode = SET then
    let
      $u_1 = I.\mathrm{op1}$,
      $(\mathrm{arr}, A) = H(u_1)$,          % $(u_r, (\mathrm{arr}, A)) \in H$
      $(\mathrm{int}, i) = I.\mathrm{op2}$,          % $i$ must be an integer
      $v = I.\mathrm{op3}$,                   % $v \in$ Scalars $\cup$ U
                                                % where the uids must be of records, arrays, oneofs.
      $t = A(i)$,                             % $t = (u_x, (\mathrm{ecq}, Q))$
      $Act', EIS' =$
         $SendToDestinations(Act, EIS, u_{FA}, \{((\mathrm{unconditional}, d_1, opnum_1), \alpha_1), \ldots,$
                             $((\mathrm{unconditional}, d_n, opnum_n), \alpha_n)\})$
    in
    $Act'$,                      % the new set of activations.
    $AddToHeap(DeleteFromHeap(H, u_1, (\mathrm{arr}, A)), u_1, (\mathrm{arr}, A'))$,
                       % the new heap reflects the fact that the ith element
                       % of the array with uid u has value v.
    $(EIS - \{(u_{FA}, k_{FA})\}) \cup Q$
                % The new EIS' includes the instructions whose addresses were in the
                % EC-queue. $(u_{FA}, k_{FA})$ is the address of the current instruction,
                % which is removed from the set of enabled instructions. When the value
                % becomes available, this instruction will be added back to the set of enabled
                % instructions.
    endlet
endif

      where $A'(j) = A(j)$,      $j \neq i$
                   $= v$,           $j = i$.
```

The use of EC-elements in data structures permits the construction of a data structure before the values of all the components have been computed. Suspensions allow demand driven computation. A suspension is a tagged triple — a tag **susp**, the uid u of some function activation, and i the index of an instruction in the activation. The instruction SETSUSP takes three arguments : an array A, an integer v_1 which is an index of the array, and another integer v_2 which is the index of an instruction in the template of the SETSUSP instruction. SETSUSP sets the v_1th element of the array to a suspension of the form $(\mathbf{susp}, u_{FA}, v_2)$ where u_{FA} is the uid of the activation template of the SETSUSP. When a SELECT tries to access the element, the suspension is replaced by an EC-queue which contains the address of the SELECT and a signal is sent to the instruction whose address is found in the suspension. The graph is so arranged that the arrival of the signal enables the instruction (u_{FA}, v_2), which initiates the computation of the value of the element.

```
if $I$.opcode = SETSUSP then
    let
     $u_1 = I.\mathrm{op1}$,
     $(\mathrm{arr}, A) = H(u_1)$,                  % $(u_1, (\mathrm{arr}, A)) \in H$
     $(\mathrm{int}, v_1) = I.\mathrm{op2}$               % $(v_1$ must be an integer
     $(\mathrm{int}, v_2) = I.\mathrm{op3}$               % $(v_2$ must be an integer
     $(u', (\mathrm{ecq}, Q) = A(i)$          % $A(i)$ must be an early-completion queue.
     $Act', EIS' =$
        $SendToDestinations(Act, EIS, u_{FA}, \{((\mathrm{unconditional}, d_1, \mathrm{signal}), \mathrm{signal}), ....,$
                              $((\mathrm{unconditional}, d_n, \mathrm{signal}), \mathrm{signal})\})$
    in
     if $|Q| = 0$ then
        % put a suspension in the ith element of the structure with uid $u_I$
      $Act'$,
      $AddToHeap(DeleteFromHeap(H, u_1, (\mathrm{arr}, A), u_1, (\mathrm{arr}, A'))),$
      $EIS' - \{(u_{FA}, k_{FA})\}$

        where $A'(j) =$ (if $j \neq v_1$ then $A(j)$ else $(\mathrm{susp}, (u_{FA}, v_2))$
     else
        % just send a signal to the instruction whose index is $v_2$
      let $Act'', EIS'' =$
        $SendResult(Act', EIS', u_{FA}, (\mathrm{unconditional}, v_2, \mathrm{signal}), \mathrm{signal})$
      in
        $Act''$,
        $H$,
        $EIS'' - \{(u_{FA}, k_{FA})\}$
      endlet
     endif
    endlet
endif
```

The SWITCH operator is used to implement the conditional graph schema. It takes two operands – a value v_1 and a boolean b. The condition fields of the destinations of the SWITCH operator must have values either **true** or **false**. If b is true, then v_1 is sent to the destinations which are marked **true**, otherwise they are sent to the destinations marked **false**. The destinations (**true**, d_{t1}, $opnum_{t1}$), ..., (**true**, d_{tp}, $opnum_{tp}$) denote the destinations to which the first operand must be sent if the second operand is true; otherwise the first operand is sent to the destinations (**false**, d_{f1}, $opnum_{f1}$), ..., (**false**, d_{fq}, $opnum_{fq}$).

```
if
$I$.opcode = SWITCH then
    let
      $u_1 = I$.op1,
      (bool, $b$) = $I$.op2,            % $b$ must be a boolean value.
      (true, $d_{t1}$, $opnum_{t1}$), ..., (true, $d_{tp}$, $opnum_{tp}$),
      (false, $d_{f1}$, $opnum_{f1}$), ..., (false, $d_{fq}$, $opnum_{fq}$) = $I$.destlist,
      $Act'$, $EIS'$ =
          if $b$ then
              $SendToDestinations(Act, EIS, u_{FA}, \{((dc_{t1}, d_{t1}, opnum_{t1}), \alpha_{t1}), ...,$
                              $((dc_{tp}, d_{tp}, opnum_{tp}), \alpha_{tp})\})$
          else
              $SendToDestinations(Act, EIS, u_{FA}, \{((dc_{f1}, d_{f1}, opnum_{f1}), \alpha_{f1}), ...,$
                              $((dc_{fq}, d_{fq}, opnum_{fq}), \alpha_{fq})\})$
          endif
    in
      $Act'$,
      $H$,
      $EIS' - \{(u_{FA}, k_{FA})\}$
    endlet
endif
```

The SWITCH-SIGNAL operator is very similar. It takes a boolean operand; if the operand has true value then it sends signals to the destinations tagged **true**, otherwise it sends signals to the destinations marked **false**.

The SIGNAL instruction requires no operands; it becomes enabled when it receives a signal on each of the signal arcs incident on it. The result of executing the instruction is a signal which is sent to the destinations listed in the **dests** field of the instruction. The state transition rule for the SIGNAL instruction is very simple and will not be specified here.

```
if $I$.opcode = APPLY then
    let
    (clsr, $C$) = $H(I$.op1),                 % $(u_1$, (clsr, $C)) \in H$
    $u_2$ = $I$.op2,                % $(u_2$, (rec, $R)) \in H$
    $F$ = $H(C($FunctionToApply)),      % $(u_f$, (fn, $F)) \in H$, $u_f = C($FunctionToApply)
    $u'$ = a new uid from U,
    $Act'$ = $AddToAct(Act, u', F)$,
    $Act''$, $EIS'$ =
        $SendResult$
         ($SendResult$
          ($SendResult$
            ($Act'$, $EIS$, $u'$, (unconditional, 1, op1), (value, $u_1$)),    % function closure.
            $u'$, (unconditional, 2, op2), (value, $u_2$)),           % arguments
            $u'$, (unconditional, 3, op1), (value, $I$.destlist))          % return link.

    in
        $Act''$, $H$, $EIS' - \{(u_{FA}, k_{FA})\}$
    endlet
endif.
```

The function closure, argument structure and the return link are sent to the first operand of the first, second and third instruction in the activation template of the called function. A RETURN instruction in the called function will send the result of the function application to the destinations of APPLY.

The TAILAPPLY instruction is used whenever tail-recursive function application is performed. It sends the closure, argument structure and the return link to the first operand of the first three instructions in the called activation. It then sends a signal to each of its destinations.

```
if $I$.opcode = TAILAPPLY then
    let
     (clsr, $C$) = $H(I$.op1),        % $(u_1$, (clsr, $C)) \in H$, the closure
     $u_2 = I$.op2,            % $(u_2$, (rec, $R)) \in H$, the arguments
     $u_3 = I$.op3,            % $(u_3$, (dests, Destinations) $\in H$, the return link
     $F = H(C$(FunctionToApply)),
                % $(u_f$, (fn, $F)) \in H$, $u_f = C$(FunctionToApply)
     $u'$ = a new uid from U,
    $Act' = AddToAct(Act, u', F)$,
     $Act'', EIS' =$
        SendResult
         (SendResult
          (SendResult
            ($Act', EIS, u'$, (unconditional, 1, op1), (value, $u_1$)),    % function closure.
                      $u'$, (unconditional, 2, op2), (value, $u_2$)),    % arguments
                      $u'$, (unconditional, 3, op1), (value, $u_3$)),    % return link.
     $Act''', EIS'' =$
        SendToDestinations($Act, EIS, u_{FA}$, {((unconditional, $d_1$, signal), signal), ...,
                          ((unconditional, $d_n$, signal), signal)})
    in
     $Act''', H, EIS'' - \{(u_{FA}, k_{FA})\}$
    endlet
endif
```

STREAM-TAILAPPLY is another instruction for function application and is used for tail-recursive evaluation of streams. It takes three arguments – a function closure which contains the stream producer, a record which will contain the next element of the stream, and the argument record. Its semantics are very similar to that of TAILAPPLY and it sends signals to the destinations listed in its dests field. The compilation of functions which generate streams and use this instruction is discussed in chapter 5.

The RETURN instruction is responsible for sending the result of a function activation to the caller. It requires two operands – a return link and a value. It constructs the addresses of the instructions which are to receive the result of the function invocation from the return link and sends the value to each of those destinations. In addition, it sends signals to those instructions in its own activation whose indices appear in the destination list of RETURN.

```
if opcode = RETURN then
    let
        $\{(dc'_1, d'_1, opnum'_1), \ldots, (dc'_p, d'_p, opnum'_p)\} = H(I.\mathbf{op1})$,
            % the list of destinations to which the value computed
            % by the function must be forwarded
        $u_v = I.\mathbf{op2}$,                % the value to be returned
        $Act', EIS_* =$
            $SendToDestinations(Act, EIS, u_{FA}, \{((\mathbf{unconditional}, d_1, \mathbf{signal}), \mathbf{signal}), \ldots,$
                        $((\mathbf{unconditional}, d_n, \mathbf{signal}), \mathbf{signal})\})$,
        $Act'', EIS'' =$
            $SendToDestinations(Act, EIS, u_{FA}, \{((\mathbf{unconditional}, d'_1, opnum'_1), \alpha_1), \ldots,$
                        $((\mathbf{unconditional}, d'_p, opnum'_p), \alpha_p)\})$
    in
            $Act''$,
            $H$,
            $EIS'' - \{(u_{FA}, k_{FA})\}$
    endlet
endif
```

The data flow graph of a function is arranged so that RELEASE is the last instruction to be enabled and executed in the activation. The effect of the instruction is to remove the activation it belongs to from the set of current activations in the system. In a "real" system, this would amount to the release of the storage occupied by the activation template to the pool of free storage in the system.

```
if opcode = RELEASE then
    $DeleteFromAct(Act, u_{FA}, FA)$
    $H$,
    $EIS - \{(u_{FA}, k_{FA})\}$
        where $FA$ is the activation template associated with $u_{FA}$.
endif
```

2.5 Summary

In the preceding section a formal specification of VIM was given by defining its operational semantics. A brief description of the functional language VIMVAL was given, and some example programs illustrated some features of the language. Some of the key features of VIM were described informally. Next, we developed a formal model of the abstract machine for VIM. An interpreter for executing the data flow instructions was defined. The operational semantics of the data flow instructions of the machine were presented. The state transition rules for operations on early-completion structures and suspensions were formalised.

This operational model is called L1 and is the specification of VIM. The next chapter defines the operational semantics of L2, in which the machine L1 is refined to include a model for storage on which [illegible]

Chapter Three

Operational Semantics of VIM with Storage

The VIM computer system is envisioned to have a hierarchically organized physical storage consisting of main-memory and a disk. Information is brought from the disk into the main memory upon demand. Since the system has finite main storage, objects are periodically moved from the main memory to the disk to create free space in the main store into which objects from the disk are brought in. It is desired that the only information brought into the main memory from the disk are those required by the computation and no other.

The address space of the system is governed by the size of the disk. For the purpose of this thesis it is assumed that a sufficiently large disk is available, thus avoiding the complications of managing the disk space. To facilitate data transfers between the disk and the main store, the address space is divided into equal sized pages, where a page is a set of contiguous words in the address space. Objects are stored in these pages. If the page size is large, it may be necessary to pack multiple objects in a page to reduce internal fragmentation. It is likely that when a page containing an object referenced by the computation is brought into main memory, objects which are not required by the computation would also be brought in since they share the same page. This conflicts with our stated goal of reading in only referenced objects from the disk into the main memory.

In VIM the basic unit of storage allocation and for disk↔main store transfer is a *chunk*. A chunk is a small page, and is expected to consist of 24-32 words. Each chunk has a unique chunk identifier (*cid*). Since the pages are small, it is unnecessary to allocate multiple objects on the same chunk. Objects which are larger than one chunk are stored in data structures made of chunks.

The small page size in VIM allows us to allocate at most one object per chunk without causing significant wastage of storage space due to internal fragmentation. When an object residing on the disk is referenced, only that object (or part of the object, if it consists of many chunks) is brought into the disk. Since large data structures are composed of many chunks, choice of a suitable data structure organization should permit large amount of sharing of information; this sharing is essential for efficient execution of structure operations in an applicative environment.

In L1 we saw that data structures reside on the heap and only the pointers to the structures (their uids) are passed among instructions. In this chapter the operational semantics of an extension of the model L1 is presented. The extended model, called L2, adds the notion of storage to L1 – data structures (arrays, records and oneofs) are stored in chunks. The storage representation of arrays in terms of chunks will be described; the concepts presented may be extended to the storage representation of records and oneofs.

The modelling of arrays as stored values makes it necessary for us to consider the issues of storage allocation and reclamation. The unit of storage allocation in L2 is a chunk. For the purpose of this thesis it is assumed that there is a large pool of *free chunks* (which are not part of any data structure). The allocation of a storage unit amounts to selecting a *cid* from this free pool and using the storage corresponding to that *cid*. In L2, chunks which are modelled as being in the main store are tagged **accessible** and the chunks which cooorespond to those resident in the disk are tagged **inaccessible**. It is assumed that the free *cid* selected corresponds to a chunk whose storage part is in the main memory (*i.e.* the chunk is tagged **accessible**).

A program executing in L2 exhibits dynamically changing storage requirements during the course of the computation. This variable demand arises from the fact that data structures are created and discarded (in the sense that they are not used any more) during program execution. The storage that is discarded can be reused for storing other data structures.

The function of a storage management scheme in a language implementation is two-fold – allocation of memory when the computation demands, and the reclamation of storage which contain the values of to discarded information structures. Reclamation of storage and its subsequent recycling allows the system to satisfy the storage requirements of the program within the existing bounds of the system, even though the total amount of memory (number of free chunks) requested by the computation far exceeds the total storage capacity of the system. There are two principal strategies for storage reclamation – the mark-and-sweep scheme and the reference-count scheme.

Mark and sweep garbage collection is the most common method of automatic storage reclamation. In a simple mark and sweep scheme, the size of the inaccessible information occupying the address space keeps growing until there is no free storage left, at which point all normal processing is suspended. All the units of storage which are in use are marked by tracing down all the structures which are accessible from the current state of the system. Then the entire memory is scanned to identify all the unmarked storage units – these are the discarded memory units which are aggregrated into the pool of free chunks and used for reallocation. A drawback of this strategy is that if the address space is very large and the physical store spans disks, the process of garbage collection can be very expensive. Simple implementations of mark-and-sweep suspend all computations other than those for reclamation; a large number of disk accesses will imply that all other computations will remain suspended for a long time. Various algorithms have been proposed which permit concurrent execution of the mark-and-sweep reclamation and other computational activities (also known as real-time garbage collection); however, they are complicated and exhibit questionable performance in real-life situations. Real-time garbage collectors of acceptable performance are difficult to implement even on single-processor systems; how the schemes may be extended to perform garbage collection with acceptable efficiency in a multiprocessor architecture remains an open problem.

The notion of using reference counts on the information structures used by the computation has been around for a long time; however, there are only a scant number of garbage collectors which use reference counts exclusively. The basic idea is very simple. A counter is associated with each information unit; it keeps a count of the number of references to the structure in the system. The counter is incremented whenever a new reference is created and decremented when one is destroyed. When the count becomes zero, the structure becomes inaccessible from the computation and the storage occupied by it may be reclaimed for reallocation. This simple scheme has one major drawback which has prevented its use in any practical garbage collector so far – it cannot reclaim circular structures. However, it has been shown that memory reclamation using reference counts is possible in the presence of certain classes of circular structures [7, 17].

Circular structures can be created only if there are operations which cause side-effect. All operations in VIMVAL are free from side effects and so the user cannot create circular structures. The creation of circular structures by the interpreter (for whatever may be the purpose) has been precluded by design[5]. These features of the VIM system make it feasible to use a reference count mechanism for garbage collection. The principal argument against reference counts is that the cost of updating the count every time a structure is manipulated may be unacceptably high. A significant advantage of reference count mechanism for garbage collection is that storage reclamation is done concurrently with the computation. Also, the scheme appears to be more amenable to implementation on a multiprocessor architecture.

In L2, two counts – **refcnt** and **setcnt** are associated with each chunk. **refcnt** contains the count of the number of references to the chunk in the current state of the machine. A chunk is reclaimed when its **refcnt** field becomes zero. The **refcnt** fields of all the structures which are pointed at by the chunk whose **refcnt** field becomes zero are also decremented. The **setcnt** field of a chunk is used only if it is the root chunk of some structure. It keeps a count of the number of elements in the structure which are EC-queues.

Some simplifications (of "real" life behaviour of computer systems) have been made in developing the formal model L2 to reduce the complexity of the model within manageable limits. In a "real" system, information which is in main memory is immediately accessible; in L2 the chunks which correspond to those resident in the main store are tagged **accessible**. The chunks which correspond to those on the disk are tagged **inaccessible**. Instructions are tagged **executable** if they have received all their operands and signals. Only such an instruction is chosen for execution. The instruction executes ("runs to completion") only if all the chunks that it requires to access are tagged **accessible**. If such is not the case then the system requests that the chunk be made accessible and the instruction is tagged **dormant**. Some other **executable** instruction is then selected and run for execution. Eventually, the requested chunk becomes accessible and the instruction which requested the chunk is made **executable**.

Data flow graphs usually expose a high degree of concurrency in most programs. It is hoped that by using suitable program transformation techniques, the number of enabled instructions at any time during the execution of the program will be very large. The overlap between instruction execution and

[5] In an implementation for his combinator reduction language, Turner [37] uses circular structures to model recursion.

disk accesses is the principal source of concurrency in the system. I expect that in an actual implementation the paging algorithms and instruction scheduling can be so ordered that the system seldom has to wait for a disk access to complete before it can execute an instruction. This translates into the following caveat – there must be at least one executable instruction during most of the time of the program execution.

3.1 Arrays and VIM-trees

We now examine a special kind of data structure called a VIM-tree, which is used to store the elements of an array in L2.

A *positional k-tree* is a directed tree with the following property : Each edge out of a node v is associated with one of the numbers in $\{0, 1, ..., k\text{-}1\}$; different edges, out of v, are associated with different numbers. It follows that the number of edges out of a node is at most k, but may be less; in fact, a leaf has none.

We associate with each leaf node v in a positional k-tree V the word consisting of the sequence of numbers associated with the edges on the path from the root r to the node v. This sequence is called the *index word* of node v. The index word also represents an integer in base k. The height of the tree is the length of longest index word in the tree [15].

A VIM-tree is a positional k-tree in which every node is associated with a chunk. The chunk associated with the root node is called the *root chunk*. A chunk has two parts – *header* and *chunkstore*. The header part of the chunk contains some book-keeping information about the chunk; the chunkstore contains the actual data values (or pointers to other chunks). Elements of an array are stored in the chunkstore part of some of the chunks associated with the leaf nodes. Those leaf chunks which contain elements of the array are called *value chunks*. All value chunks have index words of the same length. For convenience, the terms "value chunk" and "root chunk" will be used in place of "chunk associated with the leaf node" and the chunk associated with the root node". A word in the chunkstore of a value chunk in a VIM-tree is uniquely identified by the base-k integer wi, where w is the index word of the value chunk and i is the word number in the chunkstore. I shall use the terminology "the with word in the VIM-tree V" to denote the ith word in the chunk with index word w in V; wi is the *word number* (in base-k) for that element of the array in V. All the words of the chunkstore part of a chunk contain the value *unallocated* by default. If an arc from node A to B be marked i, then the ith word of chunk associated with A contains the *cid* of the chunk at B. If there is no node corresponding to the ith edge from A, the ith word of A contains the value *unallocated*.

The **refcnt** field of a chunk is a count of the number of occurrences of the *cid* of the chunk in the current state of the machine. This is used for reference counted automatic storage reclamation. The **setcnt** field of the root chunk contains the number of elements of the array which are currently EC-queues. This field is used to determine if an APPEND operation should be performed. The **depth** field of the root chunk has value d, where the height of the tree is d-1. It is used to construct the index word to the value chunk. The number of array elements that can be stored in a full VIM-tree of height d-1 is k^d.

The **lo** and **hi** fields of the root chunk of VIM-tree V contain the low and high indices of the array whose elements are stored in the tree. If the low index of the array is p and the word number for the pth element in V is $s_{d-1}s_{d-2}...s_1s_0$ then the $\mathbf{m}_{min}$ field of the root chunk is $(p - s_{d-1}s_{d-2}...s_1s_0)$, all arithmetic being done in base-k. The $\mathbf{m}_{min}$ field is used to determine the index word of the *Value* chunk in which the element which is to be accessed resides.

Let an array A with index bounds p and q be stored in a VIM tree V in which the $\mathbf{m}_{min}$ and **depth** fields of the root chunk have values m_{min} and d. For every i within the bounds, $(i - m_{min})$ is the word number corresponding to the ith element of the array A. The word corresponding to the word number wi in V is said to be *shared* if there is a chunk corresponding to the the index word w, and the **refcnt** field of some chunk along the path from the root determined by the word index has value greater than 1. Otherwise, the word is *unshared*.

3.2 Operational semantics of L2

In 1.1 we saw that arrays are represented as abstract mathematical entities – as functions mapping integers to values. In 1.2 we augment 1.1 by introducing a storage model for arrays. This chapter focuses on the operations on arrays in 1.2. The operations on records and oneofs are very similar to those on arrays; records and oneofs may be regarded as fixed-length arrays from the point of view of the implementation of their operations in the machine. Functions, activations, early-completions queues, etc. are still regarded as abstract mathematical entities. This simplifies the presentation and keeps the complexity of the model within reasonable bounds. The techniques illustrated in this thesis may be used to develop a model in which the aforesaid entities are also data structures, rather than sets and functions.

The rest of this chapter is a description of the model L2. The notation used is the same as the one used in presenting L1. Sets are denoted by bold font, elements of sets are denoted by italicised letters and names are written in a special font. Thus, **This** is a set, *This* ∈ **This** and This is a name.

A *State S* in L2 is a four-tuple consisting of *Act* (the current set of function activations), *H* (the heap), *EIS* (the set of enabled instructions) and *C* (the set of chunks which are currently in use to store the arrays on the heap).

VIM = <*Interp*, *State*> where
State = **Act** × **H** × **EIS** × **C**

Act = **U** → [**N** → **Instruction**]

$$\mathbf{H} = \mathbf{U} \rightarrow [(\{\mathtt{fn}\} \times \mathbf{Function}) \cup (\{\mathtt{ecq}\} \times \mathbf{ECQ}) \cup \{(\mathtt{instr}\} \times \mathbf{Instruction}) \cup (\{\mathtt{dests}\} \times \mathbf{Dests}) \cup (\{\mathtt{clsr}\} \times \mathbf{Clsr}) \cup (\{\mathtt{arr}\} \times \mathbf{Structure}]$$

Cid = the set of unique names of chunks.

The functions *AddToHeap*, *DeleteFromHeap*, *AddToAct* and *DeleteFromAct* are the same as defined in L1.

A structure is defined by a *cid* and a collection of chunks which store the values of the elements of the structure; the *cid* corresponds to that of the root chunk of the VIM-tree.
Structure = **Cid** × **C**

The set of enabled instructions is partitioned into two subsets depending on the tags on the instructions. When an instruction first becomes enabled it is tagged executable and added to the set of enabled instructions. Instructions with executable tags are tagged dormant if they attempt to access chunks which are inaccessible. dormant instructions become executable when the chunks they were trying to access become accessible.

EIS = $\mathfrak{P}$(EI)
EI = Status × U × N
Status = {*executable*} ∪ (**dormant** × Cid)

Each chunk has a unique name, a tag, and some storage called chunkstore. There is a unique chunkstore associated with each *cid*. The storage part consists of some header and k words for values.

C = $\mathcal{P}$(Chunk)
Chunk = Cid × {*accessible*, *inaccessible*} × **Chunkstore**
ChunkStore = **Header** × **DataPart**
Header = **Int**^{6}
DataPart = (**Scalar** ∪ U ∪ SUSP ∪ **Cid** ∪ {*unallocated*})^k
Both **Header** and **DataPart** are sets of ordered tuples.

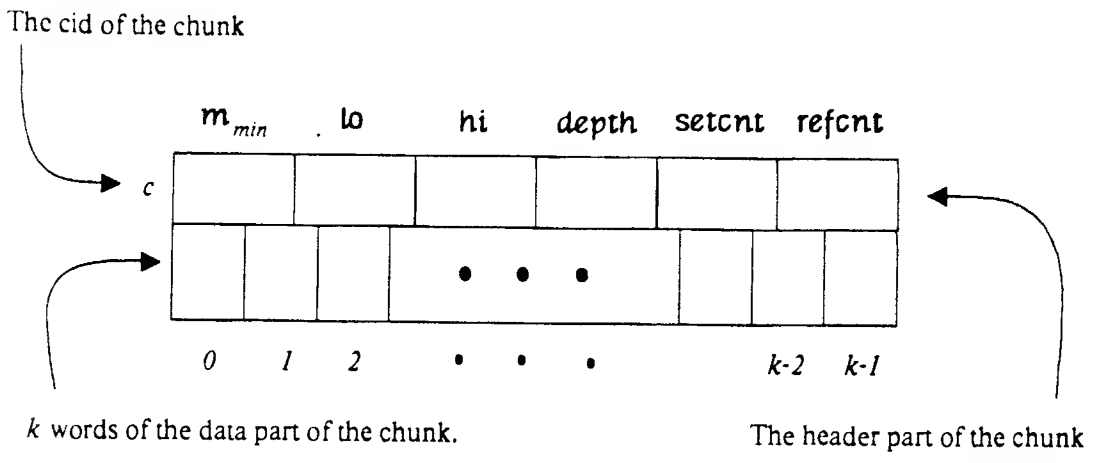
The following notation is strictly adhered to in the rest of the presentation. Ch_i always represents an element of **Chunk**, c_i denotes the unique name of the chunk Ch_i and cs_i denotes the chunkstore part of the chunk Ch_i. Thus, if Ch_w is a chunk, then c_w is its *cid* and cs_w is the chunkstore part of the chunk. c_w.$\mathbf{m}_{min}$, c_w.**lo**, c_w.**hi**, c_w.**depth**, c_w.**refcnt** and c_w.**setcnt**) denote the first six elements of the ordered tuple cs_w. $c_w[i]$ denotes the $(i+6)$th element in the tuple. The notation for drawing a chunk and specifying the contents of the chunkstore part is given in figure 9.

The *chunkgraph* of an array A in L2 is defined as follows. Let V be the VIM-tree in which the elements of the array A are stored. The nodes in the chunk graph correspond to the chunks associated with the nodes in V; if the edge from node A to B in V is marked i, then an arc is drawn from the box number i of the chunk associated with A to the chunk at B. Chunkgraphs are an efficient and concise notational convenience for specifying the operations on VIM-trees.

The *chunkgraph of a heap* is the collection of the chunkgraphs of the structures on the heap. The chunkgraph of a heap provides an easy way of indicating the sharing of chunks among structures.

Scalar values are as defined in L1.

Scalars = Integers ∪ Reals ∪ Booleans ∪ Character ∪ Null
Integers = {**int**} × ({*undef*} ∪ the set of all integers)
Reals = {**real**} × ({*undef*} ∪ the set of all reals)
Booleans = {**bool**} × ({*true*, *false*, *undef*})
Character = {**char**} × ({*undef*} ∪ the set of characters in the machine.)
Null = {**null**} × {*nil*, *undef*}


The cid of the chunk
m_{min}
lo
hi
depth
setcnt
refcnt
c
0
1
2
• • •
k-2
k-1
k words of the data part of the chunk.
The header part of the chunk

where c_1 is the *cid* of the chunk and cs_1 is the chunkstore part of the chunk.

Figure 9: Notation for drawing a chunk and specifying its value. The chunkstore part of the chunk is represented by a box, which is divided into two rows of boxes. The upper row is divided into six boxes, one corresponding to each of m_{min}, **lo**, **hi**, **depth**, **setcnt** and **refcnt** fields of the header part of the chunk, in that order. The second row contains the k words of the data part of the chunk. The unique name (*cid*) of the chunk is indicated at the left of the box. An unsigned integer in the **refcnt** or **setcnt** field means that the field now has that value. A signed integer in the box implies that the new contents of the box is equal to the old contents added to the signed integer. This notation will be used to specify how the reference counts on chunks is incremented and decremented.

ECE = U × N
ECQ = $\mathfrak{R}$(ECE)
SUSP = [U × N]

The definitions of the sets **Instruction, Function, ECE, ECQ and Dests** are the same as in L1.

Instruction = OPS × (U ∪ Scalars)^3 × N^2 × U
Dests = $\mathfrak{R}$(D)
D = {**unconditional**, **true**, **false**} × N × {**op1**, **op2**, **op3**}
Clsr = [({**FunctionToApply**} ∪ M) → U]
where C(**FunctionToApply**) ∈ U × ({**fn**} × **Function**), C ∈ Clsr

The function *SendResult* which is invoked to dispatch the result of an instruction to the destination instructions is almost identical to the one in L1. The only difference between is that in L2 the instructions which become enabled are tagged **executable**; in L1 no tagging is done. The function *SendToDestinations* is the same as in L1.

$$SendResult : \text{Act} \times \text{EIS} \times \text{U} \times \text{D} \times [[\{\textbf{value}\} \times (\text{U} \cup \text{Scalars})] \cup \{\textbf{signal}\}] \rightarrow \text{Act} \times \text{EIS}$$

```
Define $SendResult(Act, EIS, u_{FA}, (dc, i, opnum), result) \equiv$

let
    $FA = Act(u_{FA})$,
    $I = FA(i)$
in
    $AddToAct(DeleteFromAct(Act, u_{FA}, FA), u_{FA}, FA')$          % new set of activations
    if $(I'.\textbf{opcnt} = 0) \wedge (I'.\textbf{sigcnt} = 0)$ then $EIS \cup \{(\textbf{executable}, u_{FA}, i)\}$
    else $EIS$
    endif
endlet
where
    $FA'(j) = FA(j), \quad j \neq i.$
           $= I', \quad j = i.$
and   $I' \in$ Instruction,
    $I'.\textbf{op1}$ = if $opnum \neq \textbf{op1}$ then $I.\textbf{op1}$ else $t$ where $result = (\textbf{value}, t)$
    $I'.\textbf{op2}$ = if $opnum \neq \textbf{op2}$ then $I.\textbf{op2}$ else $t$ where $result = (\textbf{value}, t)$
    $I'.\textbf{op3}$ = if $opnum \neq \textbf{op3}$ then $I.\textbf{op3}$ else $t$ where $result = (\textbf{value}, t)$
    $I'.\textbf{opcnt}$ = if $opnum \in \{\textbf{op1}, \textbf{op2}, \textbf{op3}\}$ then $(I.\textbf{opcnt} - 1)$ else $I.\textbf{opcnt}$
    $I'.\textbf{sigcnt}$ = if $opnum = \textbf{signal}$ then $(I.\textbf{sigcnt}-1)$ else $I.\textbf{sigcnt}$
    $I'.destlist = I.destlist$
endfun.
```

The function *Interp* maps a state and an enabled instruction to a new state. The enabled instruction chosen by *Choice* must have tag **executable**.

$$Interp : \text{State} \times Choice(\text{EIS}) \rightarrow \text{State}$$
$$\text{where State} = \text{Act} \times \text{H} \times \text{EIS} \times \text{C}$$

A practical implementation of VIM would have a finite amount of main store and a very large amount of storage space on the disk. The contents of a chunk cannot be read unless it is present in the main memory. Chunks are read into the main memory from the disk on demand. Eventually there may not be any free storage in the main memory into which chunks may be brought in. The system frees main storage by moving some chunks from the main memory to the disk and declaring the main storage that was occupied by them to be free. New chunks from the disk are placed in this free storage, which is then marked as occupied. Effectively, chunks which are resident in the disk are not directly accessible to the computation. Thus chunks become accessible/inaccessible during the execution of a program, depending on whether they move from the disk to main memory, or from main store to disk.

Chunks which are modelled as being in the main memory are tagged **accessible**, otherwise they are tagged **inaccessible**. To capture this notion of chunks becoming inaccessible during program execution, a function *PageOut* is introduced. *PageOut* selects some chunks in the current state of the machine which have tag **accessible** and marks them as **inaccessible**.

PageOut : $\mathbf{C} \rightarrow \mathbf{C}$

Every element marked **dormant** in the set *EIS* ∈ EIS contains a pointer to a chunk (the *cid* of the chunk); the tag on the chunk is **inaccessible**. *Fetch* selects some such *cid*, tags it as **accessible** and all the instructions which had become **dormant** trying to access this chunk are tagged **executable**. The action performed by *Fetch* corresponds to the conventional notion of pages being brought into the main memory from the disk.

Fetch : $\mathbf{EIS} \times \mathbf{C} \rightarrow \mathbf{EIS} \times \mathbf{C}$

Let c be the *cid* of a chunk which is tagged **inaccessible** and let $W = \{(u, k) : (\textbf{dormant}, c, (u, k)) \in EIS\}$, $W \neq \{\}$, in some state (Act, H, EIS, C) of the machine. *Fetch*(EIS, C) returns a new set of enabled instructions given by $(EIS \cup \{(\textbf{executable}, u, k) : (u, k) \in W\}) - W$, and a new set of chunks specified by $(C \cup \{c, \textbf{accessible}, cs\}) - \{(c, \textbf{inaccessible}, cs)\}$ where cs is the chunkstore associated with the chunk with name c.

The main loop of the machine is defined by the following tail-recursive function. The machine executes an instruction, makes some chunks inaccessible and then makes some of the demanded chunks accessible.

```
Define $MainLoop(S : State) \equiv$
    let
      $(Act_1, H_1, EIS_1, C_1) = S$
    in
      if $EIS = \{\}$ then halt
      else
        let
          $e = Choice(EIS)$ where $e = (\textbf{executable}, u_{FA}, k_{FA})$,
          $EIS, C = Fetch(EIS_1, PageOut(C_1))$,
          $H = H_1$,
          $Act = Act_1$
        in
          $MainLoop(Interp(Act, H, EIS, C), e)$
        endlet
      endif
    endlet;
```

Interp defines the manner in which the state transitions are made, depending on the instruction which is being executed.

```
Define $Interp((Act, H, EIS, C), e) \equiv$
  let
    $(status, u, k) = e$,
                % The instruction must have tag executable.
    $FA = Act(u)$,
    $I = FA(k)$
  in

    if $I$.opcode = IADD then ...
    elsif $I$.opcode = MKINTARRAY then ...
      ...
    elsif $I$.opcode = APPLY then ...
      ...
    endif,
    endif
  endlet
```

The notation

$(Act, H, EIS, C) \vdash (Act', H', EIS', C')$ on e

denotes that if the state of the machine given by (Act, H, EIS, C) is the argument to *MainLoop* then (Act', H', EIS', C') is the result of executing the instruction e chosen by *Choice*, and invoking *PageOut* and *Fetch* in sequence.

We are now equipped to describe the actions of the interpreter. The scalar operations do not affect the heap and so the C component of the state is unaffected. The state transition rule of the instruction is almost identical to that in L1, the only difference being the introduction of the fourth component in the state (which remains unchanged).

```
if $I$.opcode = IADD then
    let
     (int, $m$) = $I$.op1,
     (int, $n$) = $I$.op2,
     $sum = m+n$,
     $Act'$, $EIS'$ =
            $SendResult($
             $SendResult(...($
              $SendResult(Act, EIS, (u_{FA}$, (unconditional, $d_1, opnum_1)), \alpha_1), ... ) ...,$
                              $(u_{FA}$, (unconditional, $d_n, opnum_n)), \alpha_n))$
    in
     $Act'$,
     $H$,
     $EIS'$ - {(executable, $u_{FA}, k_{FA})\}$,
     $C$
    endlet
endif
```

where $\alpha_i \in$ {(value, sum), signal}

Other scalar operations have very similar state transition rules and do not affect the heap or the set of chunks.

Operations on arrays will now be described. Figures are used to explain the algorithms for building and manipulating the trees of chunks that store the contents of the arrays.

The state resulting from the execution of the MKINTARRAY instruction is described below.

```
if $I$.opcode = MKINTARRAY then
  let
    (int, $p$) = $I$.op1,
    (int, $q$) = $I$.op2,
    $n$ = |$I$.dests|,
    ($c_1$, accessible, $cs_1$) = a new chunk,
    $u$ = a new uid,
    $A$ = ($u$, (arr, $c_1$, {($c_1$, accessible, $cs_1$)})),
    $Act'$, $EIS'$ =
      $SendToDestinations(Act, EIS, u_{FA}$, {((unconditional, $d_1$, $opnum_1$), $\alpha_1$), ...,
                ((unconditional, $d_n$, $opnum_n$), $\alpha_n$)})
  in
    $Act'$,
    $AddToHeap(H, u, A)$,
    $EIS'$ - {(executable, $u_{FA}$, $k_{FA}$)},
    $C \cup$ {($c_1$, accessible, $cs_1$)}
  endlet
endif
```

where α_i is either (value, u) or signal.

As mentioned earlier, it is assumed that an accessible free chunk is available. The contents of the chunk with *cid* c_1 is in figure 10. The contents of this chunk along with the definition of the SELECT operation ensure that any SELECT operation on this tree produces the an undefined value.

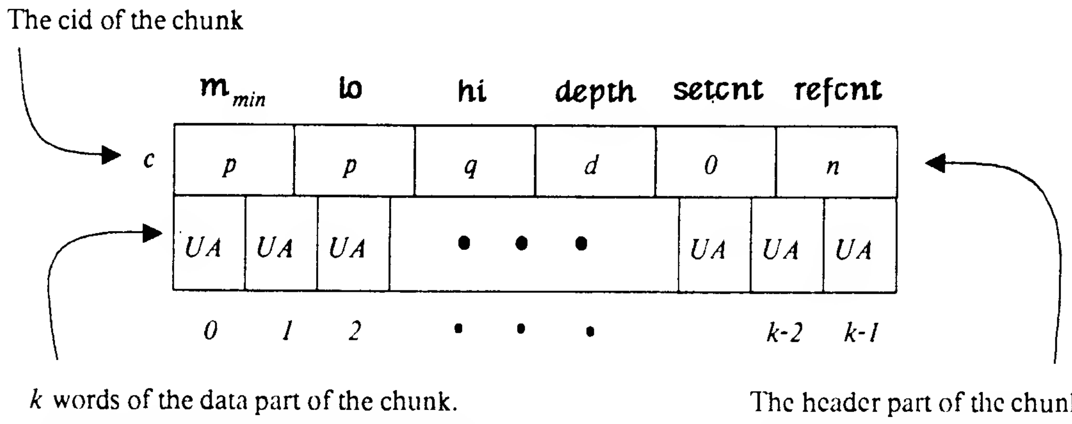


Figure 10: The chunk structure created by the MKINTARRAY instruction. The entire tree tree of chunks is not created; only the root chunk is allocated. The symbol *UA* stands for *unallocated*. The depth field is set to the value $d = \lceil \log_k(q - p + 1) \rceil$)

Let us now consider the effect of the instruction MKINTARRAYEC on the state of the machine. It acquires a collection of chunks which are not members of the set C in the current state of the machine. These chunks are used to build a tree which is illustrated in figure 11.

```
if $I$.opcode = MKINTARRAYEC then
  let
    (int, $p$) = $I$.op1,
    (int, $q$) = $I$.op2,
    $n = |I.\text{dests}|$,
    $M = q-p+1$,
    $N = M(k^{d+1}-1)/((k-1)k^{d+1})$,
    $\{(c_1, \text{accessible}, cs_1), \ldots, (c_N, \text{accessible}, cs_N)\}$ = $N$ free accessible chunks,
    $u$ = a new uid,
    $A = (u, (\text{arr}, c_1, \{(c_1, \text{accessible}, cs_1), \ldots, (c_N, \text{accessible}, cs_N)\}))$,
    $Act', EIS'$ =
      $SendToDestinations(Act, EIS, u_{FA}, \{((\text{unconditional}, d_1, opnum_1), \alpha_1), \ldots,$
                $((\text{unconditional}, d_n, opnum_n), \alpha_n)\})$
  in
    $Act'$,
    $AddToHeap(H, u, A)$,
    $C \cup \{(c_1, \text{accessible}, cs_1), \ldots, (c_N, \text{accessible}, cs_N)\}$
  endlet
endif
```

where α_i is either **signal** or (**value**, u), the contents of the chunks is shown in figure 11 and the resulting heap H' is specified by augmenting the chunkgraph of the heap H by the chunkgraph shown in figure 11.

The APPEND operation is by far the most complex operation. It requires three arguments — an array, an integer index and a value (uid of some other structure or a scalar value). Recall that the APPEND operation creates a new array only if its argument **array** (first operand) does not have any EC-queues; otherwise it is attempted for reexecution at some later time. In L2 the **setcnt** field of the root chunk of the first operand (which must be an array) contains number of EC-queues in the structure. If the **setcnt** field is not zero, then no change occurs in the state. Instead, some other instruction is selected by the *Choice* function in the next iteration of *MainLoop*. The APPEND instruction remains in the set of enabled instructions and will be eventually executed, when all the EC-queues have been replaced by values by SET instructions.

If the first operand to APPEND is a shared structure (many pointers to it exist in the current state

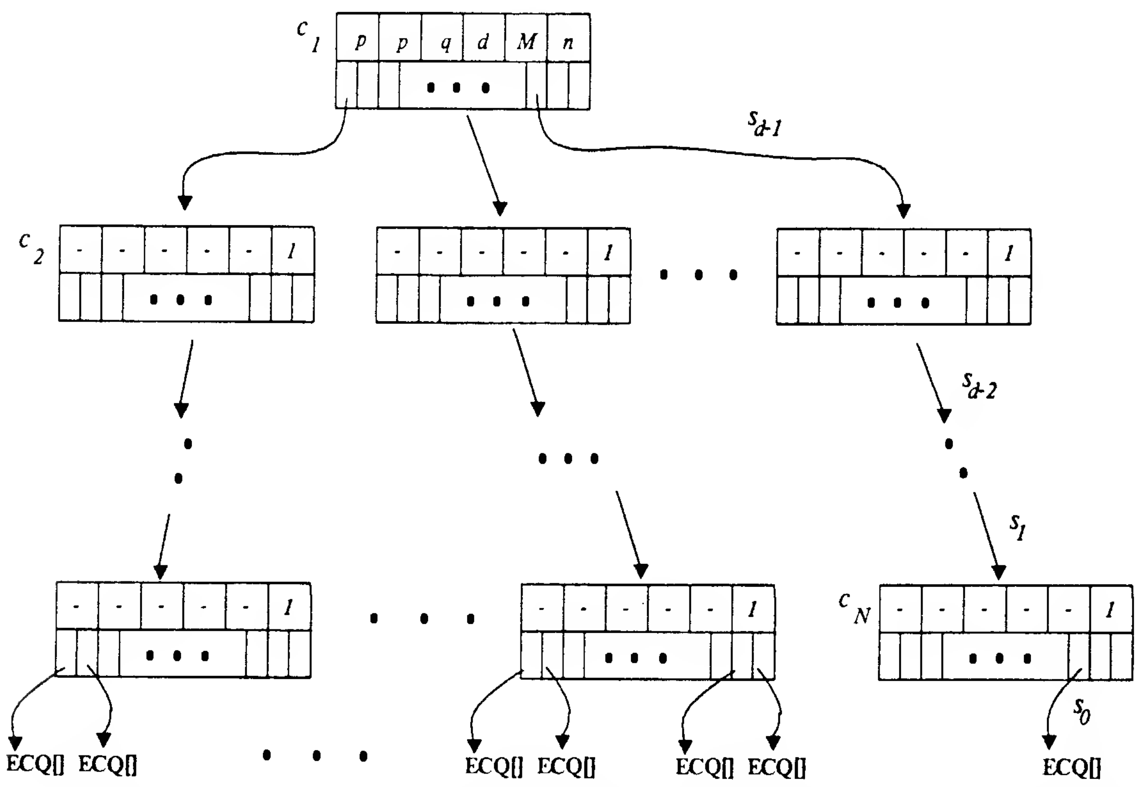


Figure 11: Tree of chunks created by created by MKINTARRAYEC. p and q are the bounds of the array, $M = q-p+1$, $d = \lceil \log_k(M \rceil)$ and n is the number of destinations of the instruction c(mkintarrayec). ecq[] denotes an empty EC-queue. Thus all the elements of the array point to empty EC-queues. $s_{d-1}s_{d-2}...s_1s_0$ is the word number of element with index q in the array. The word number of the element with index p is 00...0.

of the system), then a new array is created which shares a number of chunks with the argument array. If the value of the **refcnt** field of the root chunk of the first operand to APPEND is one, then it is possible to perform an in place update. This condition under which no copying need be performed at all to implement the APPEND operation can be taken advantage of by the compiler. If the integer index (the second operand) is outside the bounds of the argument array, APPEND creates a larger array. The transition rules are specified in the following pages.

Let the array on which the APPEND instruction is to performed have the chunkgraph shown in Figure 12.

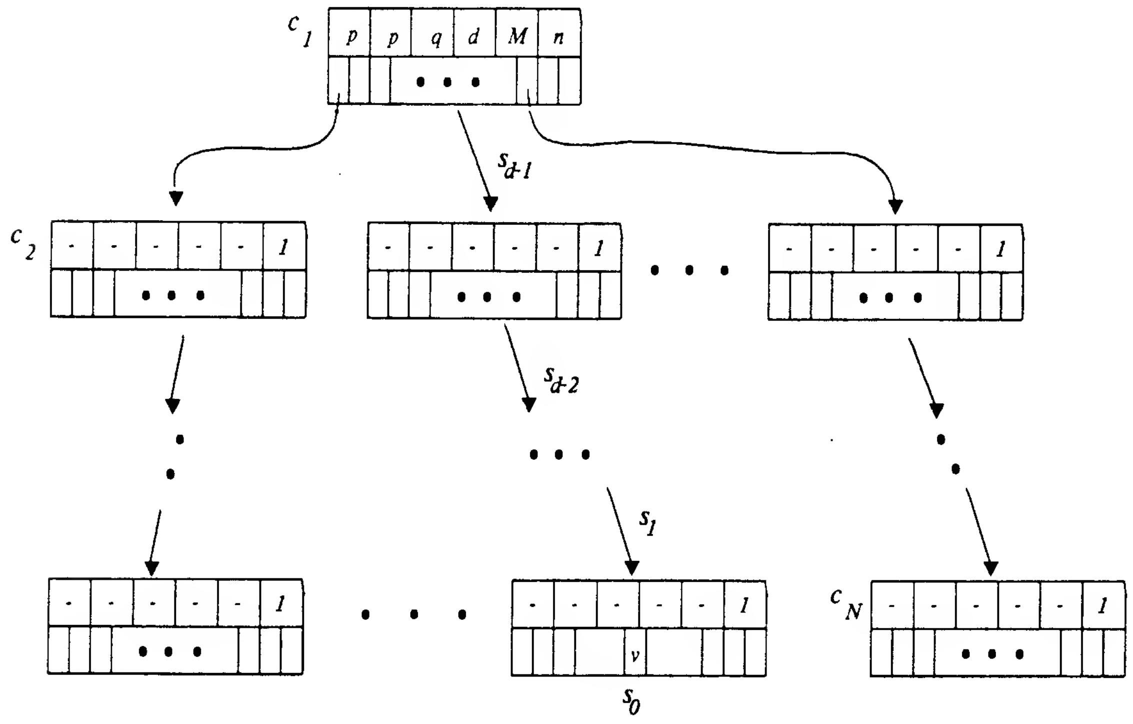


Figure 12: Chunkgraph of the array on which the APPEND operation is performed.

```
if $l$.opcode = APPEND then
  let
    $u_1 = l$.op1,
    (arr, $c_1$, $\{(c_1, af_1, cs_1), ..., (c_N, af_N, cs_N\}$)) = $H(u_1)$,
    (int, $i$) = $l$.op2,
    $x = l$.op3,
    $p = c_1$.lo,
    $q = c_1$.hi,
    $d = c_1$.depth,
    $n = |l$.dests$|$,
    $u_2$ = new uid from U
  in
    if |$i$ : element with word index $i$ of the array is a suspension
            or an EC-queue| > 0
      then Act, H, EIS, C
        % There are EC-queues in the structure — APPEND does not execute.
    elsif $p \leq i \leq q$ then
      if (word number ($i - c_1.\mathbf{m}_{min}$) in VIM-structure with root $c_1$ is unshared)
      then ...
```

```
        else ...                  % *The element is shared.*
        endif
      elsif $c_1.\mathrm{m}_{min} < i < p$ then ...
      elsif $i < c_1.\mathrm{m}_{min}$ then ...
      elsif $q < i < c_1.\mathrm{m}_{min} + k^d$ then ...
      else ...              % $i > c_r.\mathrm{m}_{min} + k^d$ ...
      endif
    endlet
  endif
```

First consider the case in which the *i*th element of the array is not shared among any other structures. The APPEND instruction performs an in place update by replacing the value of the *i*th element of the array by the new value *x*. The VIM-tree represents a new array, which is reflected by the new uid that is associated with the structure. The set of chunks *C* remains unchanged from the previous state.

If any chunk with *cid c* whose contents need to be accessed during the execution of the APPEND instruction is found to have an **inaccessible** tag, the state $S_f = (Act_f, H_f, EIS_f, C_f)$ resulting from the execution of the APPEND instruction is defined below.

$$Act_f = Act$$
$$H_f = H$$
$$EIS_f = (EIS \cup \{((\textbf{dormant}, c), u_{FA}, k_{FA})\} - \{(\textbf{executable}, u_{FA}, k_{FA})\}$$
$$C_f = C$$

The instruction thus remains in the set of enabled instructions. Eventually when the chunk *c* becomes accessible (caused by the function *Fetch*), it will again become executable.

The actions of the interpreter for the APPEND operation will now be presented. Consider first the case when the *i*th element of the array on which the APPEND is performed is an unshared element and *i*th lies within the bounds of the array[6].

[6] A simplifying assumption made here is that there is a leaf chunk corresponding to the *i*th element of the array. If that is not the case then new chunks are added to the set of chunks in the state and the set *C* is augmented suitably.

```
if $p \leq i \leq q$ and (word number $(i - c_1.\mathbf{m}_{min})$ in VIM-structure with root $c_1$ is unshared)
then
  let
    $Act', EIS' =$
        $SendResult($
          $SendResult(...($
            $SendResult(Act, EIS, (u_{FA}, (\textbf{unconditional}, d_1, opnum_1)), \alpha_1), ... ) ...,$
                        $(u_{FA}, (\textbf{unconditional}, d_n, opnum_n)), \alpha_n))$
  in
    $Act',$
    $(H - (u_1, (\textbf{arr}, c_1, \{(c_1, \textbf{accessible}, cs_1), ..., (c_N, \textbf{accessible}, cs_N)\})))$
                $\cup \{(u_2, (\textbf{arr}, c_1, \{(c_1, \textbf{accessible}, cs_1), ..., (c_N, \textbf{accessible}, cs_N\}))\}$
    $EIS' - \{(\textbf{executable}, u_{FA}, k_{FA})\},$
    $C'$ where $C'$ reflects the fact that an in situ change has been made to the chunks of
                the structure $A$ as shown in figure 13.
  endlet
endif
```

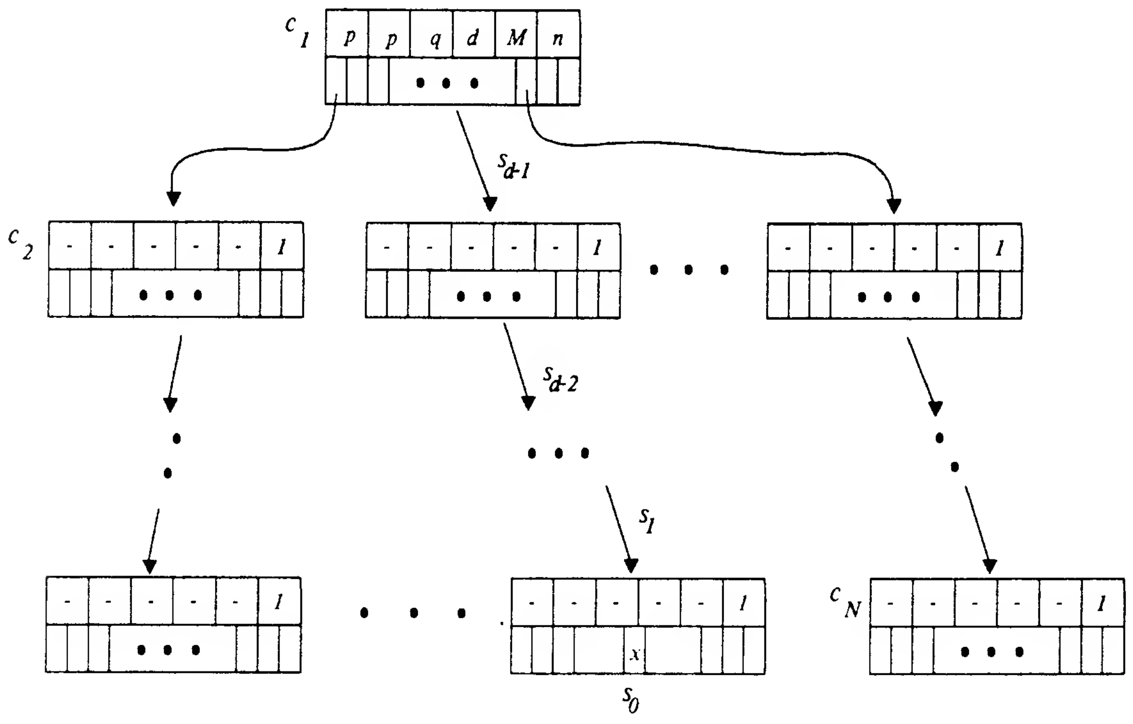


Figure 13: Chunkgraph of the result of APPEND on the array shown in figure 12 when $p \leq i \leq q$ and the element is an unshared element ($M = 0$ and $n = 1$).

Now consider the case in which the element is shared among structures on the heap. A new structure is created by copying chunks along the access path. The reference count of the argument structure A is decremented by 1. If A.**refcnt** becomes zero, the node is deleted from the heap and the reference counts of all the chunks that c_1 points at are also decremented. This may cause a cascade of deletion of nodes from the heap. If any of the chunks whose contents needs to be accessed (for decrementing its reference count field or for reading the contents of its chunkstore) is tagged **inaccessible** then the instruction is made dormant and no changes are made to the *Act*, *H* and *C* components of the state. The state transition rule for such a case has already been described.

```
if $p \leq i \leq q$ and (word number $(i - c_1.\mathbf{m}_{min})$ in VIM-structure with root $c_1$ is shared)
then
  let
    $Act', EIS' =$
        $SendResult($
         $SendResult(...($
          $SendResult(Act, EIS, (u_{FA}, (\textbf{unconditional}, d_1, opnum_1)), \alpha_1), ... ) ...,$
                        $(u_{FA}, (\textbf{unconditional}, d_n, opnum_n)), \alpha_n)),$
    $(c'_1, \textbf{accessible}, cs'_1), ..., (c'_{d-1}, \textbf{accessible}, cs'_1)$ = new chunks unused in $C$,
    $H' = H \cup \{u_2, (\textbf{arr}, c'_1,$ {chunks shared between argument and result arrays}
            $\cup\{(c'_1, \textbf{accessible}, cs'_1), ..., (c'_N, \textbf{accessible}, cs'_N)\}$
            $-$ {structures whose root chunk has refcnt field has value zero}
  in
    $Act'$,
    $H'$,
    $EIS' - \{\textbf{executable}, u_{FA}, k_{FA})\}$,
    $C \cup \{(c'_1, \textbf{accessible}, cs'_1), ..., (c'_N, \textbf{accessible}, cs'_N)\}$
        - {chunks whose refcnt field becomes zero}
  endlet
endif
```

Now consider the case in which $A.\mathbf{m}_{min} < i < A.\mathbf{lo}$. The resulting VIM structure is of the same height as the old structure. New chunks are acquired from the pool of free chunks and are initialized such that much of the information common to the two arrays is shared on chunks. A.**refcnt** is decremented: if it becomes zero then A is deleted from the heap, as are structures whose reference count becomes zero. The chunks in which the elements of the structures are stored are deleted from C if their **refcnt** field becomes zero. It is easily seen that the result structure A' preserves the semantics of the append operation — a SELECT done on A' on L2 maps to the same value as a SELECT on the result of APPEND(A, i, x) in L1.

The third case arises when $c_1 \leq i < c_1.\mathbf{lo}$. The resulting tree has the same height as the argument

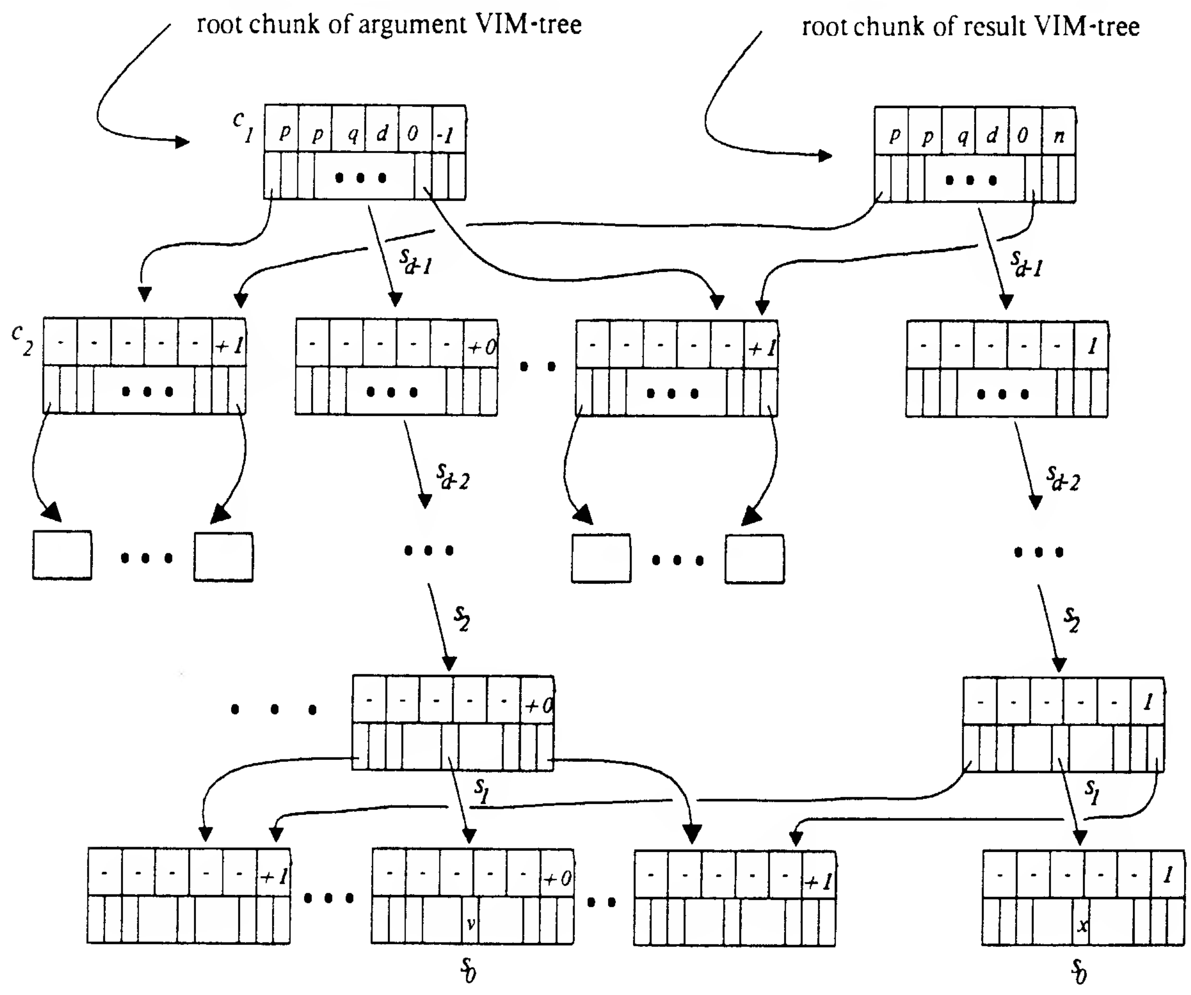


Figure 14: Chunkgraph produced by APPEND on the *i*th element of the structure whose chunkgraph is shown in figure 12. This is the case when $p \leq i \leq q$ and the element is a shared element.

tree. Chunks along the access path are copied, while others are shared as shown in figure 16. The **lo** field of the root chunk of the VIM-tree for the result array is set to *i*. No change is made to the value represented by the argument structure. The **refcnt** field of c_1 is decremented. If that becomes zero, then the usual process of decrementing reference counts is started. The deletion of structures must be reflected in the resulting heap and set of chunks in the state.

```
if $c_1.\mathrm{m}_{min} < i < p$ then
  let
    $Act', EIS' = \ldots$
  in
  ... .
  endlet
endif
```

The action of the interpreter on this occasion is similar to that in the previous cases. See figure 16 for the chunkgraph of the part of the heap which is affected.

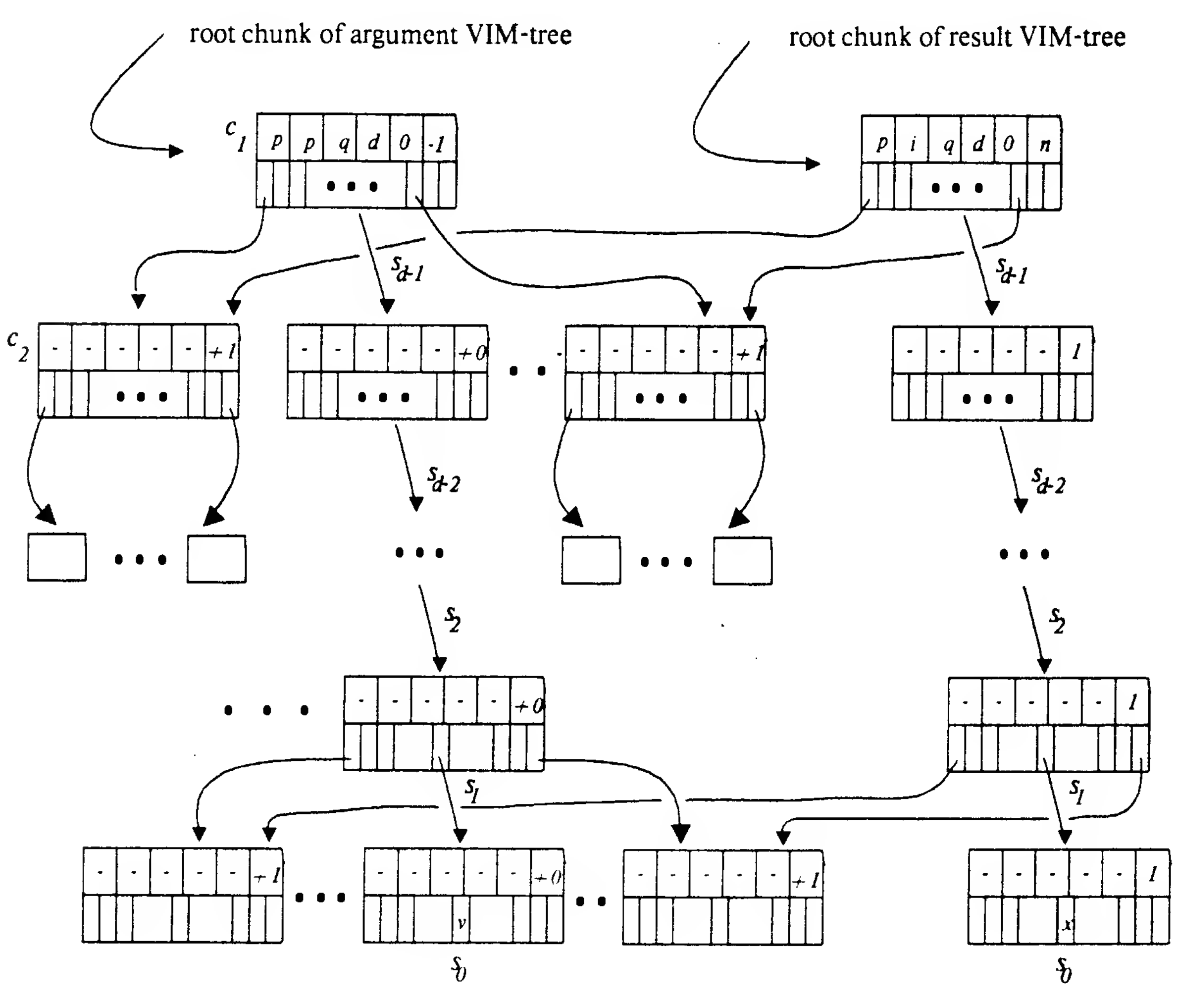

root chunk of argument VIM-tree
root chunk of result VIM-tree
c1
p p q d 0 -1
p i q d 0 n
c2
+1
+0
1
s_{d-1}
s_{d-2}
s_2
s_1
s_0
v
x

Figure 15: Chunkgraph for result of APPEND when $c_1.\mathrm{m}_{min} < i < p$.

Now we consider the case in which the height of the VIM-tree resulting from the APPEND on A is greater than the VIM-tree for A. This happens when $i < A.\mathbf{m}_{min}$. The VIM-tree for the result **array** A' created by APPEND is of the minimum such height that $A'.\mathbf{m}_{min} < i$, and SELECT(A, $A.\mathbf{m}_{min}$) = SELECT(A', $A.\mathbf{m}_{min}$). The semantics of the APPEND operation (as specified in L1) is preserved; a formal proof of this assertion will be presented in the next chapter. The reference count of A does not change since one reference to it is consumed by APPEND while a new one is inserted in A'.

Given $i < c_1.\mathbf{m}_{min}$, a tree of the minimum possible height must be constructed such that if c'_1 be the root of the resulting tree then $i\text{-}c'_1.\mathbf{m}_{min} > 0$. The appropriate height can be computed as follows.

Let d be the height of the resulting tree, which is shown in figure 17. Clearly, the word number of $c_1.\mathbf{m}_{min}$ in this tree is 100...00 such the length of the sequence is d.

$$\begin{aligned}
&\therefore c_1.\mathbf{m}_{min} - c'_1.\mathbf{m}_{min} = 100..00\\
&\Rightarrow c'_1.\mathbf{m}_{min} = c_1.\mathbf{m}_{min} - (100...00), \text{ in base } k\\
&\text{so that } c'_1.\mathbf{m}_{min} = c_1.\mathbf{m}_{min} - k^{d-1}.
\end{aligned}$$

$$\begin{aligned}
&\text{Now, } i - c'_1.\mathbf{m}_{min} \geq 0\\
&\Rightarrow i - c_1.\mathbf{m}_{min} + k^{d-1} \geq 0\\
&\Rightarrow k^{d-1} \geq c_1.\mathbf{m}_{min} - i\\
&\Rightarrow d\text{-}1 \geq \log_k(c_1.\mathbf{m}_{min} - i)\\
&\Rightarrow d \geq \log_k(c_1.\mathbf{m}_{min} - i) + 1\\
&\Rightarrow d = \lceil \log_k(c_1.\mathbf{m}_{min} - i) \rceil + 1.
\end{aligned}$$

It is of interest to note that the word number of $c_1.\mathbf{m}_{min}$ could also be ss000...00, where $1 \leq s \leq k$. The arithmetic would be vary similar to the above case.

The instruction becomes dormant if any of the required chunks is tagged **inaccessible**. The **refcnt** field of the argument array need not be decremented because even though the instruction consumes a reference to athe array, a new one is created in the new structure. Therefore, in this case no storage reclamation will be triggered.

```
if $i < c_1.\mathbf{m}_{min}$ then
  let
    $Act', EIS' = ...$
  in
  ...
  endlet
endif
```

see figure 17.

The cases for the rest of the two cases are similar to the previous two cases and are not discussed.

The state transition rule for the SELECT operation is specified below. The select operation decrements the reference count of the argument structure. If the element which is being accessed by SELECT is an EC-queue then the address of SELECT is placed on the EC-queue and the instruction is removed from *EIS*. If the element is a suspension then the suspension is replaced by an EC-queue containing the address of SELECT and a signal is sent to the instruction whose address was specified in the suspension. If the element is a scalar value then the value is simply dispatched to the destinations. If the result of SELECT is a structure, then its **refcnt** field is incremented by n, the number of destinations of SELECT. If A.**refcnt** becomes zero due to the decrementing of the reference count it is deleted from the heap. The **refcnt** fields of the chunks that the root chunk c_1 of A points at are also decremented and the chunk c_1 reclaimed. The decrementing of **refcnt** fields may trigger more reclamation. If any of the chunks which needs to be accessed is tagged **inaccessible** then SELECT is made dormant; no changes are made to the *Act*, *H* and *C* components of the state.

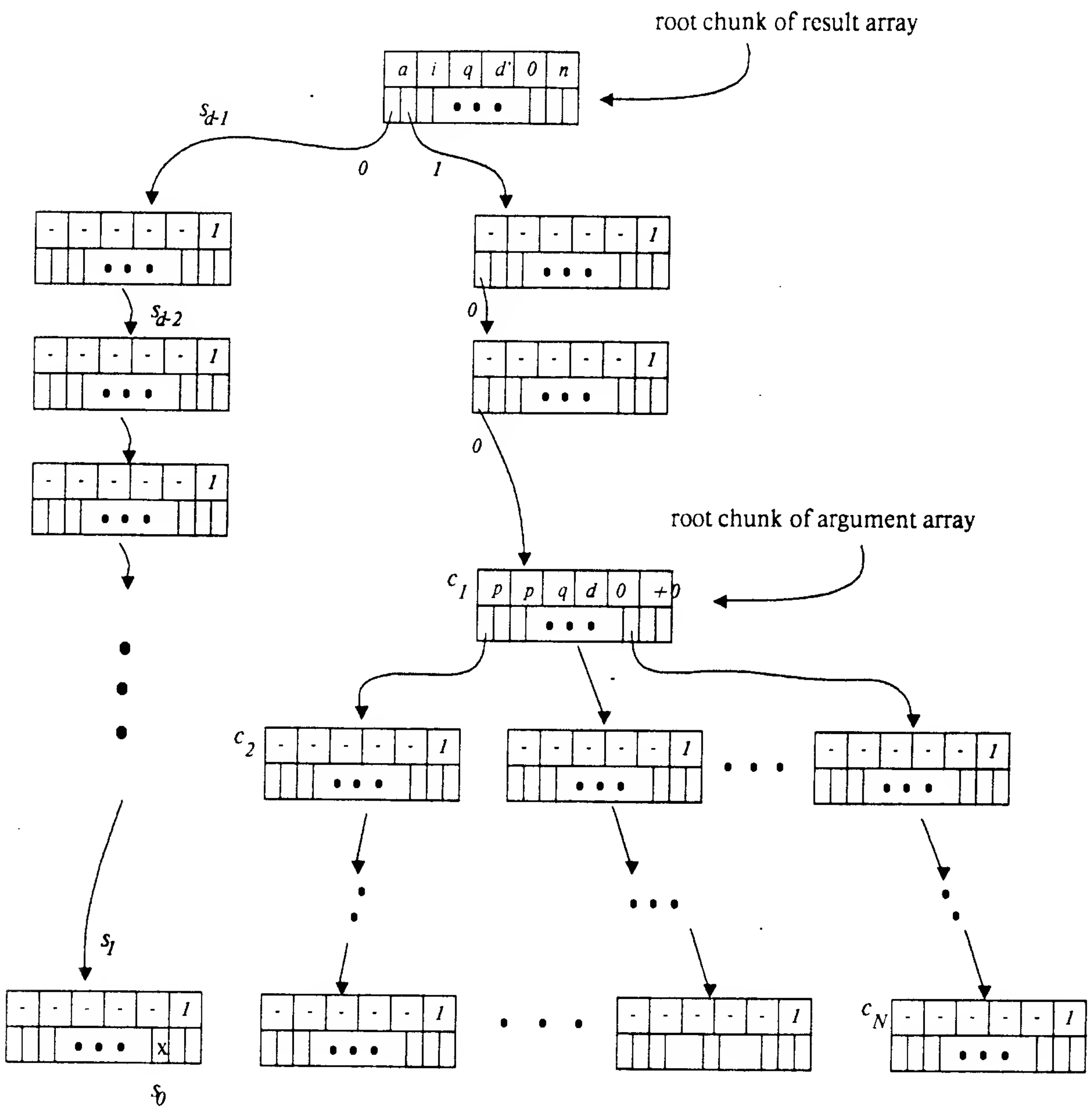


Figure 16: Chunkgraph for result of APPEND when $i < c_1.\mathbf{m}_{min}$. The height of the result VIM-tree is $d = \lceil \log_k(c_1.\mathbf{m}_{min} - i) \rceil + 1$. $a = c_1.\mathbf{m}_{min} \cdot k^{d-1}$.

```
if $I$.opcode = SELECT then
  let
    $u_1 = I.\text{op1}$,
    $(\text{arr}, c_1, \{(c_1, af_1, cs_1), \ldots, (c_N, af_N, cs_N\})) = H((u_1)$,
    $(\text{int}, i) = I.\text{op2}$,

  in
    if ($i < c_1$.lo) or ($i > c_1$.hi) then undef
    else
     let
       $x$ = contents of word number $(i - c_1.\text{m}_{min})$ in the VIM-tree with root $c_1$
     in
       if $x$ is a scalar then ...
       elsif $x$ is a uid $u_B$ corresponding to structure $B$ then ...
               % send the value and increment refcnt
               % field of the root chunk of the VIM-tree for B.
       elsif $x$ is an EC-queue then add $(u_{FA}, k_{FA})$ to the queue, etc.
       elsif $x$ is a suspension then [replace suspension by EC-queue with $(u_{FA}, k_{FA})$,
               increment $c_1$.setcnt, decrement $c_1$.refcnt, etc.]
       endif
     endlet
    endif
  endlet
endif
```

Observe that if c_1.refcnt becomes zero after SELECT executes, the node corresponding to the uid u_1 is deleted from the resulting heap and the refcnt fields of all the structures that it points to are decremented. If any chunk required by SELECT (either for accessing the element or for updating the refcnt fields) is tagged inaccessible, the instruction is made dormant, thus changing *EIS*; the other components of the state remain unaffected. When the chunk becomes accessible later, the SELECT instruction becomes executable.

The transition rule for SET is given below.

```
if $I$.opcode = SET then
  let
    ($u_1 = I$.op1,
    (arr, $c_1$, {($c_1$, $af_1$, $cs_1$), ..., ($c_N$, $af_N$, $cs_N$)}) = $H(u_1)$,
    (int, $i$) = $I$.op2,
    $x = I$.op3
  in
    if the ($i$-$c_1$.m$_{min}$)th word of the VIM-tree with root $c_1$ is an EC-queue $Q$ then
      let
        $Act'$, $EIS'$ = send signals to destinations of $e$,
        $EIS''$ = $EIS' \cup$ {(executable, $u'$, $k'$) : ($u'$, $k'$) $\in Q$},
        $H'$ = the old $H$ plus the change in the contents of the chunk of $A$ is reflected,
        $C'$ = same as $C$ except that the chunk containing the EC-queue contains $x$
                and decrement refcnt and setcnt fields of $c_1$
      in
        $Act'$, $H'$, $EIS''$, $C'$
      endlet
    else
      let
        $Act'$, $EIS'$ = send signals to destinations of $e$
      in
        $Act'$,
        $H$,
        $EIS'$,
        $C$
      endlet
    endif
  endlet
endif
```

The transition rule for SETSUSP is defined as follows.

```
if $I$.opcode = SETSUSP then
  let
    ($u_1$, (arr, $c_1$, {($c_1$, $af_1$, $cs_1$), ..., ($c_N$, $af_N$, $cs_N$)})) = $I$.op1,
    (int, $i$) = $I$.op2,
    (int, $m$) = $I$.op3
  in
    if word number ($i$-$c_1$.m$_{min}$) in VIM-tree with root $c_1$ is an empty EC-queue then
      let
        $Act'$, $EIS'$ = send signals to destinations of $e$,
        $H'$ = reflect the fact that the $i$th element of the array is a suspension
                 and the EC-queue at the element is deleted from heap
                 and elements deleted from heap due to refcnt becoming zero
        $C'$ = different from $C$ in that the chunk containing the $i$th element
                 now contains a suspension and decrement refcnt field of $c_1$
      in
        $Act'$,
        $H'$
        $EIS'$,
        $C'$
      endlet
    elsif the element is an non-empty EC-queue then
      let
        $Act'$, $EIS'$ = send signals to destinations of $e'$ and to ($u_{FA}$, $m$),
        $C'$ = decrement refcnt fields of $c_1$ and reclaim, if applicable,
        $H'$ = Reflect the changes due to changes in $C$ and due to reclamation
      in
        $Act'$, $H'$, $EIS'$, $C'$
      endlet
    else
      let
        $Act'$, $EIS'$ = send signals to destinations of $e'$,
        $C'$ = decrement refcnt fields of $c_1$ and reclaim, if applicable,
        $H'$ = Reflect the changes due to changes in $C$ and due to reclamation
      in
        $Act'$, $H'$, $EIS'$, $C'$
      endlet

    endif
  endlet
endif
```

The set of instructions related to function application (APPLY, TAILAPPLY, STREAM-TAILAPPLY, RETURN) also do not affect the heap or the set of chunks and their transition rules may thus be directly adapted from 1.1.

3.3 Discussion

The problem of efficiently implementing data structures in functional languages is of long standing. Various solutions have been proposed by researchers so that APPEND type operations can be done on such aggregrates of values without copying the entire array.

I-structures proposed by Arvind are write-once structures. I-structures solves the problem of read-before write synchronization in a concurrent system; however, the implementation described causes the structure to be copied when an append operation is done on it.

Myers proposed an implementation of applicative lists on AVL-trees [30]. The paper describes a generalization of an AVL-tree, called an AVL-dag, which is used as a representation for linear lists. He presents algorithms which oerform applicative manipulation of linear lists in time that is proportional to the logarithm of the length of the list. He also gives algorithms that perform SELECT and APPEND operations on fixed sized arrays with N elements in time $O(KN^{1/K})$, where K can be chosen arbitrarily.

Hudak and Bloss [21] have recently proposed schemes for statically inferring situations in which an APPEND may be implemented as place updates. Failing this, they propose that reference counts be maintained on the structures; the entire structure is copied if an APPEND occurs on a structure with its reference count greater than one.

None of the above solutions satisfactorily address the issue of sharing information among structures. The new data structure VIM-tree proposed in this thesis allows APPEND and SELECT operations to be performed in logarithmic time. Moreover, the algorithms common information to be shared among structures, so that the storage requirements of programs is reduced significantly. The following factors influenced the design of the data structure for representing arrays : sharing, fast SELECT and APPEND operations, and the constraint that the storage has a physical hierarchy. B-trees and AVL-trees were considered candidates for representing arrays; however, analysis indicated that the amount of processing required to balance the trees is substantially more than that for VIM-trees. It was found that balancing a k-way AVL-tree or B-tree would require a larger number of chunks (than for a VIM-tree) to be accessed, many of which might not be in the main memory. It was desirable to keep the branching factor of the trees quite high so that the depth of the tree that had to be traversed to perform the frequent SELECT operations would be quite low. For example, an array of 4096 elements can be stored in a 16-way VIM-tree of height three. The use of trees of chunks to store data structures also eliminates the problem of compaction.

This chapter also described a reference count mechanism which is used for reclamation of chunks. Reference counting permits real-time garbage collection in VIM. The operational semantics of the instructions are defined such that if a chunk required to be accessed by an instruction is not in the main memory, then the instruction is not executed. In a more detailed model, it may be possible (and maybe worthwhile in an actual implementation) to consider partial execution of instructions, so that an instruction is removed from the set of enabled instructions once it has been chosen for execution by the scheduler.

Chapter Four

Equivalence of L1 and L2

In chapter 2 the operational semantics of L1 was presented. L1 was then refined to model hierarchical physical stoge consisting of main memory and disk and a representation for data structures of the storage model was described in Chapter 3. As mentioned earlier, L1 provides the specification for any implementation of an archicture for VIM. We desire that programs executed on L1 produce the same result as that produced by running the program on L2, and vice versa. This would demonstrate that L2 indeed satisfies the specifications of L1. A formal proof of the equivalence of L1 and L2 is presented in this chapter.

Let P be a program written in the base language. Let $Translate_{L1}$ translate a program in the base language to some initial state for L1. Similarly, $Translate_{L2}$ produces an initial state in L2 for a program in the base language.

The heap of L1 is a directed, acyclic graph in which the nodes are the arrays, closures, etcc, and an arc between two nodes in the graph denotes that one structure is a component of the other. We want to capture concept of a node in the heap being accessible (not to be confused with tags **accessible**) from the current *State* of the computation. If the node is not accessible then the structure associated with the node is not usable, and is garbage.

Definition 4-1: Let $S = (Act, H, EIS)$ be a state of machine L1. A node on H in S is *reachable* if one of the following two conditions are satisfied:

1. There is some unexecuted instruction in Act which has a pointer to the node (holds the uid of the node), or
2. It is a component of some structure on H, and the node corresponding to that structure is reachable in S.

The preceding chapter gave an informal definition of the chunkgraph of a heap. The chunkgraph of a heap in some state is now formally defined in graph-theoretic terms.

Definition 4-2: Let $S' = (Act', H', EIS', C) \in$ State in L2. let $C' = \{Ch_1, Ch_2, ..., Ch_n\}$, where Ch_i is of the form (c_i, af_i, cs_i) and af_i is either **accessible** or **inaccessible**.

The *chunkgraph* of H in S is a graph $G = (V, E)$ where $V = \{Ch_1, Ch_2, \ldots, Ch_n\}$ and $E = \{(c_i, c_j) : (\exists p \in \{0, 1, \ldots, k\text{-}1\})\ c_i[p] = c_j\}$. Recall that the $c[m]$ denotes the contents of the mth word in the chunkstore part of the chunk whose *cid* is c.

It is possible that two different states in L2 represent the same set of values, the only difference being that the set of chunks used to store the elements of arrays in the two states are different. Two such states are said to be *similar*; the formal definition of the similar relation is given below.

Define two functions $StValue_1$ and $StValue_2$ for L1 and L2, respectively, as follows. If u be a uid and H be a heap in some state in L1 then $StValue_1(u, H)$ returns an ordered set; the elements of the set are the results of SELECT operations done on the array associated with u, SELECT being performed for all integer indices. If there is no structure in L1 with uid u, the result of $StValue_1(u, H)$ is the null set. $StValue_2$ is also defined in the same manner, except that the heap must be in some state in L2.

Definition 4-3: Let $S_1, S_2 \in$ State in L2, where $S_1 = (Act_1, H_1, EIS_1, C_1)$ and $S_2 = (Act_2, H_2, EIS_2, C_2)$.

S_1 and S_2 are *similar* if

1. $Act_1 = Act_2$
2. $\forall u \in U\ [StValue_2(u, H_1) = StValue_2(u, H_2)]$
3. $(\forall u \in U)(\forall k \in N)(\forall s \in \text{Status})\ [(u, k, s) \in EIS_1 \Rightarrow [(\exists s' \in \text{Status}) : (u, k, s') \in EIS_2]]$
4. The chunkgraphs for H_1 in S_1 and H_2 in S_2 are isomorphic.

The last condition simply indicates that in two similar states the chunks exhibit the same sharing relationships, except that the chunkids are different.

It is easily seen that the similar relation is an equivalence relation, and the set of similar states constitutes an equivalence class.

4.1 Proof of Equivalence of L1 and L2

Informally, two machines are equivalent if they produce identical results for a given program. Moreover, there is a correspondence between computational states that each machine goes through.

We shall use an adaptation of the McGowan mapping technique for proving the equivalence of two machines [25]. A similar technique was used by Berry [6] to prove the equivalence of two information structure models of block structured languages.

Let P be a program in the base language. Schematically, the computation of P on L1 and L2 is shown in the Figure 18. A computation is a sequence of states starting with some initial state; if the computation terminates, then the final state contains the result. $Translate_M(P)$ produces an initial state for the program P on the machine M. S_n is the final state of L1 on computation of P and $ResultValue_M(S_n)$ prints the value of the node on the heap which contains the result of the computation. Let S_0 be some initial state of machine L1. The computation of the machine L1 starting at the initial S_0 is denoted by $MainLoop_{L1}(S_0)$. Function $Final(MainLoop_M(S_0))$ gives the final state of the machine in a computation, if it halts. Similar notation is used for the machine L2. In the following discussion, rule definitions which have the same name in both L1 and L2 are distinguished by subscripted L1 or L2.

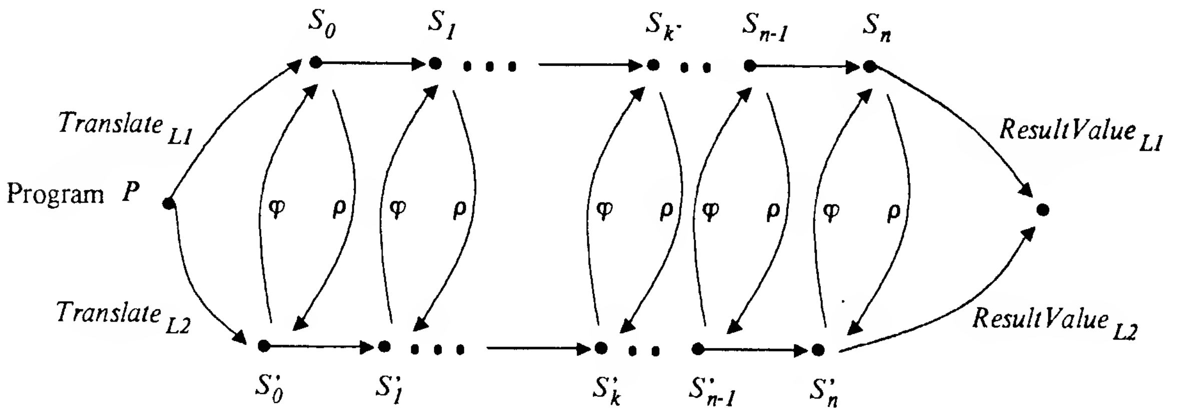


Figure 17: McGowan mapping of states of L1 and L2.

The mappings φ and ρ provide a map between the states of the machines L1 and L2 on the computation P. To show that L1 and L2 are equivalent for programs in the base language, it suffices to construct mappings φ and ρ

$$\forall S' \; \varphi : S' \rightarrow S$$
$$\forall S \; \rho : S \rightarrow S'$$

where $S' \in \textbf{State}_{L2}$ and $S \in \textbf{State}_{L1}$ such that for all programs P in the base language the following holds :

Let $S_0 = Translate_{L1}(P)$ and $S'_0 = Translate_{L2}(P)$. $MainLoop_{L1}(S_0)$ produces in succession the states $S_0, S_1, \ldots$ and $MainLoop_{L2}(S'_0)$ produces in succession $S'_0, S'_1, \ldots$

Then

1. $\rho(S_0) = S'_0$
   $\varphi(S'_0) = S_0$.
2. If $\rho(S_i) = S'_i$ and $S_i \neq Final(MainLoop_{L1}(S_0))$ then $\rho(S_{i+1}) = S'_{i+1}$.

   If $\varphi(S'_i) = S_i$ and $S'_i \neq Final(MainLoop_{L2}(S'_0))$ then $\varphi(S'_{i+1}) = S_{i+1}$.
3. If $S_n = Final(MainLoop_{L1}(S_0))$ then $\rho(S_n) = Final(MainLoop_{L2}(S'_0))$

   If $S'_n = Final(MainLoop_{L2}(S'_0))$ then $\varphi(S'_n) = Final(MainLoop_{L1}(S_0))$
4. $ResultValue_{L2}(S'_n) = ResultValue_{L1}(\varphi(S'_n))$ and $ResultValue_{L1}(S_n) = ResultValue_{L2}(\rho(S_n))$ where $S'_n = Final(MainLoop_{L2}(S'_0))$ and $S_n = Final(MainLoop_{L1}(S_0))$

Theorem 4-4: L1 is equivalent to L2 for the base language.

Proof: We exhibit McGowan mappings φ and ρ to prove the equivalence. φ maps a state S' of L2 to a corresponding state S of L1, and ρ maps a state S of L1 to a corresponding state S' of L2.

First we describe mapping $\varphi(S') = S$.

Let $S' = (Act', H', EIS', C')$.

1. Construct $Act = Act'$.

2. Construct H as follows. Let $U_{H'} = \{u : u$ is the uid of node on $H'\}$.

   $\forall\ u \in U_{H'}$ [if $H'(u) \in$ **Structure** then $(u, A) \in H$ such that $A \in$ **Array** and $StValue_{L1}(u, H) = StValue_{L2}(u, H')$
   otherwise $(u, H'(u)) \in H$]

   Clearly, if the reference counting is done correctly, then the following holds : $\forall\ u$ $|\{i : \exists u' \{u'\} \times \{\textbf{arr}\} \times \text{Array} \in H$ and $\text{SELECT}(u', i) = u\}| = A.\text{refcnt}$ where $(\textbf{arr}, A) = H'(u)$ and $A = (u, c_1, \{(c_1, af_1, cs_1), \ldots, (c_N, af_N, cs_N)\})$.

3. Construct EIS as follows.

   $EIS = \{(u, k) : (\text{Status} \times \{(u, k)\}) \in EIS'\}$

Secondly, we describe the mapping $\rho : S \rightarrow S'$ such that $S' = \rho(S)$.

Let $S = (Act, H, EIS)$.

1. Construct $Act' = Act$

2. Construct H' as follows.

   let $U_R = \{ u : u$ is the uid of a reachable node in $H\}$

   For $u \in U_R$), if u is the uid of an array then $(u, c_1, \{(c_1, af_1, cs_1), ..., (c_N, af_N, cs_N)\}) \in H'$ such that $StValue_{L2}(u, H') = StValue_{L1}(u, H)$. Also, c_1.**refcnt** = (The number of occurrences of the uid u in unexecuted instructions of Act or in reachable nodes in H) and c_1.**setcnt** = (The number of suspensions or EC-queues in the structure with uid u).

   If u is not the uid of an array (or record or oneof) then $(u, H(u) \in H'$.

3. $EIS' = \{($**executable**$, u, k) : (u, k) \in EIS\}$

4. $C' = \{(c, af, cs) : (\exists u\, \exists c_1\, \exists C : (u, c_1, C) \in H'$ and $(c, af, cs) \in C)\}$.

Observe that S' is a member of an equivalence class under the similar relation.

Thirdly, we show that φ and ρ meet the requisite conditions.

Condition 1 :
If $\varphi(S'_0) = S_0$ then $\rho(S_0) = S''_0$ where S''_0 is a member of an equivalence class under the similar relation. Also, $\varphi(S''_0) = S_0$. By judicious of chunks, we can obtain $\rho(S_0) = S'_0$.

Condition 2 :

If $\varphi(S'_i) = S_i$ and S'_i is not a final state then $\varphi(S'_{i+1}) = S_{i+1}$. We shall consider each instruction and show that the maps hold after the execution of the instruction provided the map was correct prior to the execution of the instruction.

Let $e' = ($**executable**$, u, k) \in EIS'_i$ which is executed. Assume that all the chunks needed by $Interp_{L2}$ during the execution of e' are tagged **accessible** by some omniscient *Fetch* and *PageOut* functions. Essentially, we are ensuring that an instruction in L2 does not become dormant during execution. (We shall later see that this assumption places no restriction on the generality of the result).

For ease of exposition, define three functions φ_{Act}, φ_{EIS} and φ_H such that $\varphi(S') = \varphi((Act', H', EIS')) = (\varphi_{Act}(Act'), \varphi_H(H'), \varphi_{EIS}(EIS'))$.

1. Scalar instructions :
   The result of the instruction is sent to instructions in Act'_i, the resulting activation being Act'_{i+1}.

The same set of instructions in Act_i receive the result of the scalar instruction in L1, to produce Act_{i+1}.

The actions of $SendResult_{L1}$ and $SendResult_{L2}$ on Act_i and Act'_i is the same, as is easily seen by looking at their definitions. Therefore

$\varphi_{Act}(Act'_{i+1}) = Act_{i+1}$

$EIS'_{i+1} = (EIS'_i - \{e'\}) \cup \{(\textbf{executable}, u, k) : (u, k)$ is a destination of e' and $I.\textbf{opcnt} = 0$ and $I.\textbf{sigcnt} = 0\}$, where $I = (Act'_i(u))(k)\}$.

Also, $EIS_{i+1} = (EIS_i - \{e\}) \cup \{(u, k) : (u, k)$ is a destination of e' and $I.\textbf{opcnt} = 0$ and $I.\textbf{sigcnt} = 0\}$, where $I = (Act_i(u))(k)$.

Since $\varphi_{EIS}(EIS'_i) = EIS_i$ it is clear that $\varphi_{EIS}(EIS'_{i+1}) = EIS_{i+1}$.

The heap is unaffected. Therefore $H'_{i+1} = H'_i$ and $H_{i+1} = H_i$. Since $\varphi_H(H'_i) = H_i$, it is obvious that $\varphi_H(H'_{i+1}) = H_{i+1}$.

Thus for scalar instructions the map φ holds.

2. <u>APPEND instruction</u>
By the same reasoning as used for scalars, in the state produced by APPEND, $\varphi_{Act}(Act'_{i+1}) = Act_{i+1}$, under the stated preconditions.

We have $\varphi_{H'_i} = H_i$. We will look at the different cases that arise during the execution of APPEND.

Let u_1 be the uid of the array A; u_1 is the first argument to APPEND.

a. <u>Number of EC-elements or suspensions in the array > 0</u> : $EIS'_{i+1} = EIS'_i$ in L2.

By precondition and the definition of φ and ρ, in L1, this corresponds to the case that $|\{i :$ element i of A in H_i is a suspension or an EC-queue$\}| > 1$. Therefore $EIS_{i+1} = EIS_i$. Therefore $\varphi_{EIS}(EIS'_{i+1}) = EIS_{i+1}$.

The heap is unaffected since the instruction does not execute. Thus $\varphi_H(H'_{i+1}) = H_{i+1}$ since $\varphi_H(H'_i) = H_i$.

Therefore φ holds for this case of APPEND.

b. <u>No suspensions or EC-elements and A.**refcnt** = 1.</u>
As before, $\varphi_{Act}(Act'_{i+1}) = Act_{i+1}$ and $\varphi_{EIS}(EIS'_{i+1}) = EIS_{i+1}$.

After e' executes, A is deleted and is absent in H'_{i+1}; a new array is placed on H'_{i+1}. All structures which were pointed at by A have their **refcnt** field decremented. If they are pointed at only by A then their **refcnt** will become zero and they will be absent in H'_{i+1}.

Since $\varphi_{S'_i} = S_i$ and A.**refcnt** $= 1$, A must be reachable from H_i only from e. Therefore, after e executes, A is no longer reachable in the new heap H_{i+1} and a new heap is added to this heap for the new array. All nodes in H_i which were reachable only through A will also become unreachbale in H_{i+1}.

Therefore under the given preconditions, $\varphi_H(H'_{i+1}) = H_{i+1}$.

Thus, $\varphi(S'_{i+1}) = S_{i+1}$ for this case of APPEND.

c. <u>No suspensions or EC-elements and A.**refcnt** > 1</u>
As in the previous case, $\varphi_{Act}(Act'_{i+1} = Act_{i+1}$ and $\varphi_{EIS}(EIS'_{i+1}) = EIS_{i+1}$.

After e' executes, A remains on the heap since its **refcnt** field does not become zero. A new array is added to the heap H'_{i+1} and the value of its **refcnt** field is equal to the number of destination instructions of e'.

Since $\varphi_H(H'_i) = H_i$, A is pointed at by many objects in the current *State*, and is reachable from more than one instruction or reachable structure. After e executes, A becomes unreachable in H_{i+1} through e. However, A remains reachable on the heap.

Therefore, given that $\varphi(S'_i) = S_i$, we have $\varphi_H(H'_{i+1}) = H_{i+1}$ for this case of APPEND.

This finally yields $\varphi(S'_{i+1}) = S_{i+1}$ for the APPEND instruction under the given preconditions.

3. <u>SELECT instruction</u>
Let A and j be the first and second operands of SELECT, respectively. We shall prove that φ holds for SELECT by considering the various cases that arise during the execution of SELECT.

a. <u>Element being accessed is not EC-queue or suspension</u> : By reasoning as for scalars, $\varphi_{Act}(Act'_{i+1}) = Act_{i+1}$ and $\varphi_{EIS}(EIS'_{i+1}) = EIS_{i+1}$.

In L2, if the result of SELECT is a structure B then B.**refcnt** $= n$, the number of destinations of e'. If A.**refcnt** $= 1$ then A is deleted from H'_{i+1}. If A is deleted from the resulting heap, the **refcnt** fields of the structures it points to must also be decremented.

In L1, if the result of the instruction is a astructure B, it is reachable in H_{i+1} from the instructions which are the destinations of e. If A were reachable only through e then it becomes reachableunreachable in H_{i+1}.

as doo all structures which were reachable only through A in H_i.

Thus we get that $\varphi_H(H'_{i+1}) = H_{i+1}$ showing that $\varphi(S'_{i+1}) = S_{i+1}$ for this case of SELECT.

b. Element being accessed is an EC-queue
$EIS'_{i+1} = EIS'_i - \{e'\}$.

Also, the execution of e in L1 yields $EIS_{i+1} = EIS_i - \{e\}$.

Since $\varphi_{EIS}(EIS'_i) = EIS_i$, it is obvious that $\varphi_{EIS}(EIS'_{i+1}) = EIS_{i+1}$.

$Act'_{i+1} = Act'_i = Act_i = Act_{i+1}$ so that $\varphi_{Act}(Act'_{i+1}) = Act_{i+1}$.

On executing e', the heap H'_{i+1} is the same as H'_i except that the EC-queue has an additional element (u, k). On executing e in L1, the resulting heap H_{i+1} is the same as H_i except that the EC-queue has an additional element (u, k).

Since $\varphi_H(H'_i) = H_i$, clearly it is the case that $\varphi_H(H'_{i+1}) = H_{i+1}$.

Therefore, under the given preconditions, $\varphi(S'_{i+1}) = S_{i+1}$ for this case of SELECT.

c. The element is a suspension
The execution of e' in L2 causes its removal from EIS'_{i+1}. A signal is sent to the suspended instruction whose address is found in the suspension, which may become enabled.

The actions of L1 are the same so that $\varphi_{EIS}(EIS'_{i+1}) = EIS_{i+1}$ and $\varphi_{Act}(Act'_{i+1}) = Act_{i+1}$ since $\varphi(S'_i) = S_i$.

In L2, if the ith element is a non-empty EC-queue then (u, k) is added to the EC-queue and this change is refelected in the heap H'_{i+1}. The same actions accur in L1 to yield H_{i+1}.

In L2, if the ith element is an empty EC-queue then it is replaced by an EC-queue containing only (u, k) and the EC-queue is added to the heap H'_{i+1}. The same action occurs in L1.

Therefore, given that $\varphi_H(H'_i) = H_i$, it is the case that $\varphi_H(H'_{i+1}) = H_{i+1}$.

Therefore, given the $\varphi(S'_i) = S_{i+1}$ and S'_i is not a final state, $\varphi(S'_{i+1}) = S_{i+1}$ after the SELECT instruction is executed.

4. SET instruction
Let u_1 be the uid of the array A which is the first argument of SET, and let j and x

be the second and third operands.

Signals are sent to the destinations of e', producing Act'_{i+1}. In L1, signals are set to the corresponding destinations of e in Act_i, yielding Act_{i+1}.

Since $\varphi_{Act}(Act'_i) = Act_i$, we get $\varphi_{Act}(Act'_{i+1}) = Act_{i+1}$ after SET executes.

After e' executes it is removed from EIS'_{i+1}. The addresses of the instructions in the EC-queue for the ith element of A are added to the set of enabled instructions. The instructions which receive signal from set and thus become enabled are added to the enabled instruction set, too, to yield EIS'_{i+1}. The corresponding actions occur in L1; if (u', k') is an entry in the EC-queue then (**executable**, u', k') is added to the set of enabled instructions in L1.

Therefore, given that $\varphi_{EIS}(EIS'_i) = EIS_i$, we get $\varphi_{EIS}(EIS'_{i+1}) = EIS_{i+1}$.

The heaps are affected in the following way. The EC-queue at the ith element A is replaced by x. If the ith element is an EC-queue then decrement A.**setcnt**.

The corresponding actions occur in L1 with heap H_i. The EC-queue at the ith element of A becomes unreachable in H_{i+1}. A now has one less EC-queue, if the ith element was an EC-queue.

Therefore, given that $\varphi_H(H'_i) = H_i$ we get $\varphi_H(H'_{i+1}) = H_{i+1}$.

Therefore, given the condition that $\varphi(S'_i) = S_i$ and S'_i is not the final state, $\varphi(S'_{i+1}) = S_{i+1}$ for the SET instruction.

5. SETSUSP instruction
Let the three arguments to the instruction be u_1, i and m. Let u_1 be the uid of the array A.

After e' executes, it disappears from EIS'_{i+1}. If the ith element of A is not an empty EC-queue then a signal is sent to the destination instruction (u, m). Corresponding action occurs in L1. Also, signals are sent to the destinations of e' and e in L2 and L1, respectively. Thus the corresponding set of instructions get enabled in the two machines.

$\therefore$ $\varphi(S'_i) = S_i$ implies that after setsusp executes $\varphi_{EIS}(EIS'_{i+1}) = EIS_{i+1}$ and $\varphi_{Act}(Act'_{i+1}) = Act_{i+1}$.

If the ith element of A is an empty EC-queue, then it is replaced by a suspension. The EC-queue is deleted from the heap H'_{i+1}. The same is done in L1 to produce the heap H_{i+1}.

If the ith element of A is a non-empty EC-queue then H'_{i+1} and H_{i+1} are identical to H'_i and H_i respectively.

Thus for both cases, given that $\varphi_H(H'_i) = \varphi_i$ we conclude that $\varphi_H(H'_{i+1}) = H_{i+1}$.

From the above analysis we conclude that given $\varphi_{S'_i} = S_i$ and S'_i is not the final state, $\varphi(S'_{i+1}) = S_{i+1}$ for the SET instruction.

6. MKINTARRAY and MKINTARRAYEC instructions
After e' executes it disappears from EIS'_{i+1}. The uid of the result of the instruction (or a signal) is sent to the destinations of e', which may then become enabled and so would be added to the set of enabled instructions to yield EIS'_{i+1}. Corresponding actions are performed by the L1 machine. ∴ given that $\varphi(S'_i) = S_i$ we conclude that $\varphi_{EIS}(EIS'_{i+1}) = EIS_{i+1}$ and $\varphi_{Act}(Act'_{i+1}) = Act_{i+1}$.

Suppose the instruction is MKINTARRAY. The execution of e' causes a new node corresponding to an array to be added to the heap H'_{i+1}. Any SELECT operation on the array produces an undefined value. In L1, a new node is added to the heap coresponding to an array due to the execution of e. Recall that in L1 an array is represented as a function mapping indices to values; the array added to the heap maps all indices to the undefined value.

Suppose the instruction is MKINTARRAYEC. e' adds a new node to the heap such that an element within the bounds of the array points at an empty EC-queue. Anu SELECT operation which tries to access an element outside the bounds of the array would produce an undefined value. In L1, the function is defined such that the indices within the specified bounds map to empty EC-queues and indices outside the bounds are mapped to undefined values.

Therefore, given the $\varphi_H(H'_i) = H_i$, we get $\varphi_H(H'_{i+1}) = H_{i+1}$.

Hence, given $\varphi(S'_i) = S_i$ and S'_i is not the final state, $\varphi(S'_{i+1}) = S_{i+1}$ for the MKINTARRAY and MKINTARRAYEC instructions.

7. APPLY instruction
A new function activation is added to Act'_{i+1} as a result of the execution of e'. The first three instructions of the nre activation receive the closure, argument list and return link, respectively. Corresponding action occurs in L1. ∴ under the given preconditions, $\varphi_{Act}(Act'_{i+1}) = Act_{i+1}$.

e' is deleted from EIS'_i and instructions which received the closure, argument list or return link, if they become enabled, are added to the set of enabled instructions to yield EIS'_{i+1}. L1 acts in tha same manner to yield EIS_{i+1}.

Therefore given that $\varphi_{EIS}(EIS'_i) = EIS_i$, we get $\varphi_{EIS}(EIS'_{i+1}) = EIS_{i+1}$, provided that S_i is not a final state.

The heap is unaffected, so that $H'_i = H'_{i+1}$ and $H_i = H_{i+1}$. Since we have $\varphi_H(H'_i) = H_i$, it is trivially true that $\varphi_H(H'_{i+1}) = H_{i+1}$.

$\therefore$ under the given preconditions, $\varphi(S'_{i+1}) = S_{i+1}$.

The cases for TAILAPPLY and STREAM-TAILAPPLY are similar and are left as an exercise.

8. RETURN instruction
A glance at the transition rules for the RETURN instruction in L1 and L2 tells us that there is no difference between them, except that the instructions which become enabled in L2 are tagged **executable**. The heap is unaffected. Thus we can conclude that $\varphi(S'_{i+1}) = S_i$, given that $\varphi(S'_i) = S_i$ and that S'_i is not a final state.

9. RELEASE instruction
The execution of e' deletes the activation of e' from Act'_i to produce Act'_{i+1}. Similarly, the activation of e disappers from Act_i to produce Act_{i+1}. Since $\varphi(S'_i) = S_i$, we conclude that $\varphi_{Act}(Act'_{i+1}) = Act_{i+1}$.

$EIS'_{i+1} = EIS'_i - \{e'\}$ and $EIS_{i+1} = EIS_i - \{e\}$. Therefore, under the stated preconditions, $\varphi_{EIS}(EIS'_{i+1}) = EIS_{i+1}$.

The assertions of the validity of the mapping for the heap needs some comments. The transition rule for this instruction is identical in both machines. In L1, all nodes in the heap H_i which were reachable only through instructions in the activation to which RELEASE belongs become unreachable.

The base language ensures that a RELEASE instruction is enabled only after all the instructions which have pointers to nodes corresponding to structures – arrays, records, etc. on the heap have been executed. This is done by arranging the data flow graph for a function such that the RELEASE instruction is the last instruction to become enabled. This ensures that when RELEASE is enabled, the instructions which received structure operands have already executed; if the structures were pointed at only by the instructions, then their **refcnt** field must be 1 in H'_i and would become zero after the execution of e' and would be deleted from the heap. Therefore, the mapping φ_H still holds.

Thus we conclude that if S_i is not a final state and $\varphi(S'_i) = S_i$ then after RELEASE executes, $\varphi(S'_{i+1}) = S_{i+1}$.

We demonstrated that the φ indeed provides the desired mapping. Similar reasoning may be used to show that the mapping ρ is preserved by instruction execution. The map $\rho(S)$ is a member of an equivalence class under the similar relation. This is an artifact due to the different chunks which may be used to represent the same data structure, and hence, the heap.

Thirdly, we prove that φ and ρ satisfy the third condition for equivalence. Clearly, if S_i is *Final* then $\varphi(S_i)$ is final both will have $EIS_i = EIS'_i = \{\}$. Similarly, if S'_i is final then $\rho(S'_i)$ is final.

Finally, let S_n and S'_n be the final states of machines L1 and L2, respectively. By the previous two cinditions, $\varphi(S'_n) = S_n$ and $\rho(S_n) = S'_n$, by a judicious choice of chunks.

We proved that at every step during the execution of program P the set of values represented on the heap were the same. Therefore, $ResultValue_{L1}(S_n) = ResultValue_{L2}(S'_n)$.∎

We now argue informally that even if the *Fetch* function is not omniscient, that is, it does not mark as **accessible** all the chunks required during the execution of the instruction, the result produced by the computation is the same as would be if it were omniscient. Recall that if a chunk which is required to be accessed during the execution of an instruction is marked **inaccessible**, the instruction is made dormant and no change is made to the *Act*, *H* and *C* components of the machine. The state of the machine is changed only by the fact that the status of the instruction changes in the *EIS* component of the state. Some other instruction may be selected for execution; eventually when the chunk becomes **accessible** the instruction becomes **executable**. The order of execution of instructions is thus different. Let the sequence of states during the computation of program P on L2 (in which the instructions may become dormant) be $S''_0 S''_1 \ldots S''_n$. The map φ gives us a corresponding sequence of states in L1. Since this is a non-deterministic state transition system, it can be shown that the result of this computation is the same as for the original sequence on L1, and so the result is the same.

4.2 Discussion

It was shown that L2 satisfies the specifications of VIM by proving that L1 and L2 are equivalent. This technique of taking a component of the VIM system (in this case, it is the structure memory) and designing an implementation which is formally proved to satisfy the specifications given by L1 can be extended to design the entire machine. At each step of the implementation, the resulting formal model must be proven to satisfy the specifications of L1. This methodology of designing by successive refinement with formal proofs of equivalence between the original model and its refinement will be very useful in the design of more complex implementations of VIM, such as in a computer system with multiple processing elements.

Chapter Five

A Base Language for VIM

I shall now present a base language for VIM; I shall do this by describing the base language graph corresponding to the VIMVAL language constructs. These graphs are characterised by a property called *safety*. A graph is *safe* if every instruction (in the graph) which receives a structure operand becomes enabled before the activation, of which the instruction is a part, is removed from the set of activations by the RELEASE instruction. The firing of the instruction would cause the reference counts of the the operand structures to be appropriately decremented. The discussion in chapter 4 described why the two machines are equivalent for this base language. In terms of pragmatics, the use of safe graphs would ensure that the chunks which contain garbage values are reclaimed in the actual implementation of VIM. The base language described is not the only possible one; the intent of this chapter is to give the reader a flavor of how one might go about designing the language. No formal proof will be given to demonstrate that the graphs are indeed safe and that a compiler using the base language as its target language will always generate safe graphs; the interested reader may convince himself of the safety of programs written in this base language by examining each graph and the operational semantics of each of its instructions.

5.1 The Let expression

The let construct permits local bindings to be expressed and is of the form :

let $x_1 = E_1$,
$x_2 = E_2$,
...
$x_n = E_n$
in
$E(x_1, x_2, ..., x_n)$
endlet

where the E's all represent data flow graphs.

The base language graph for the above let expression is shown in figure 19.

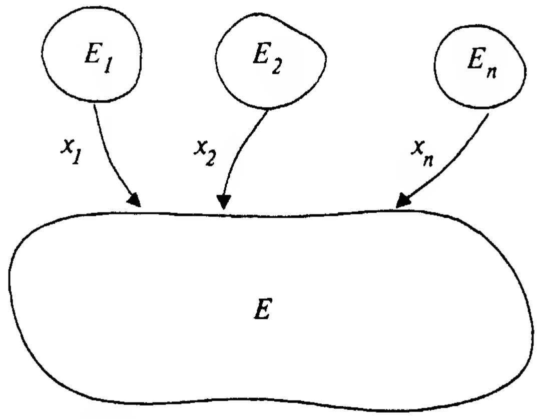


Figure 18: Translation of a let expression

5.2 Conditional Expression

The conditional expression in VIMVAL is of the form :

if f **then** $g(x_1, x_2, ..., x_n)$
else $h(x_1, x_2, ..., x_n)$

where g and h are data flow graphs and f denotes a graph that computes a boolean result. In some graphs, the values of all the operands of an instruction are known at the time of compilation. It is necessary to send a signal to such instructions to enable them. If there is such an instruction in either of graphs g and h and the instruction is not in the graph of a **let** or **tagcase** expression inside g or h, then the SWITCH-SIGNAL instruction is used; it is omitted otherwise. Each branch of the **if** expression must produce the same number of signals; this is accomplished by feeding all the signals in an arm into a SIGNAL instruction which generates one signal. If neither of the arms produces signals, then the parts of the graph that deal with them maybe omitted. If the RELEASE instruction is triggered by the signal produced by the **if** expression, and f, g and h are all safe graphs, then it is the case that the graph for the conditional expression is also safe.

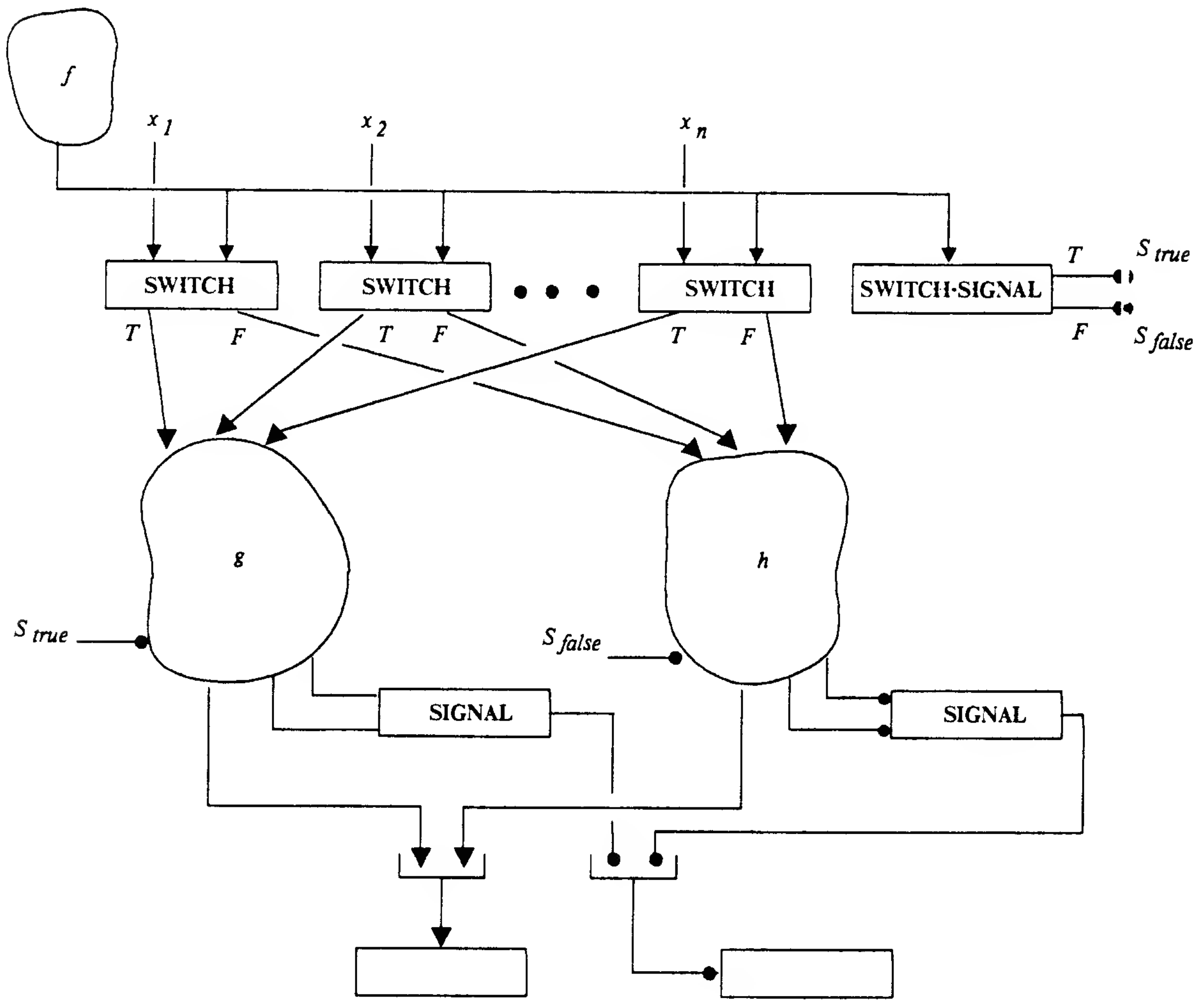


Figure 19: Translation of a conditional expression

5.3 The Tagcase Expression

Let T be a **oneof** with tags $t_1, t_2, \ldots, t_n$. The tagcase expression is :

tagcase T

  tag $t_1 : E_1(u_1, u_2, \ldots, u_a)$;
  tag $t_2 : E_2(v_1, v_2, \ldots, v_b)$;
  .
  .
  tag $t_n : E_n(z_1, z_2, \ldots, z_m)$

endtag

where the E's are data flow graphs. The use of the SWITCH-SIGNAL and SIGNAL instructions is governed by the same considerations as for the conditional expression.

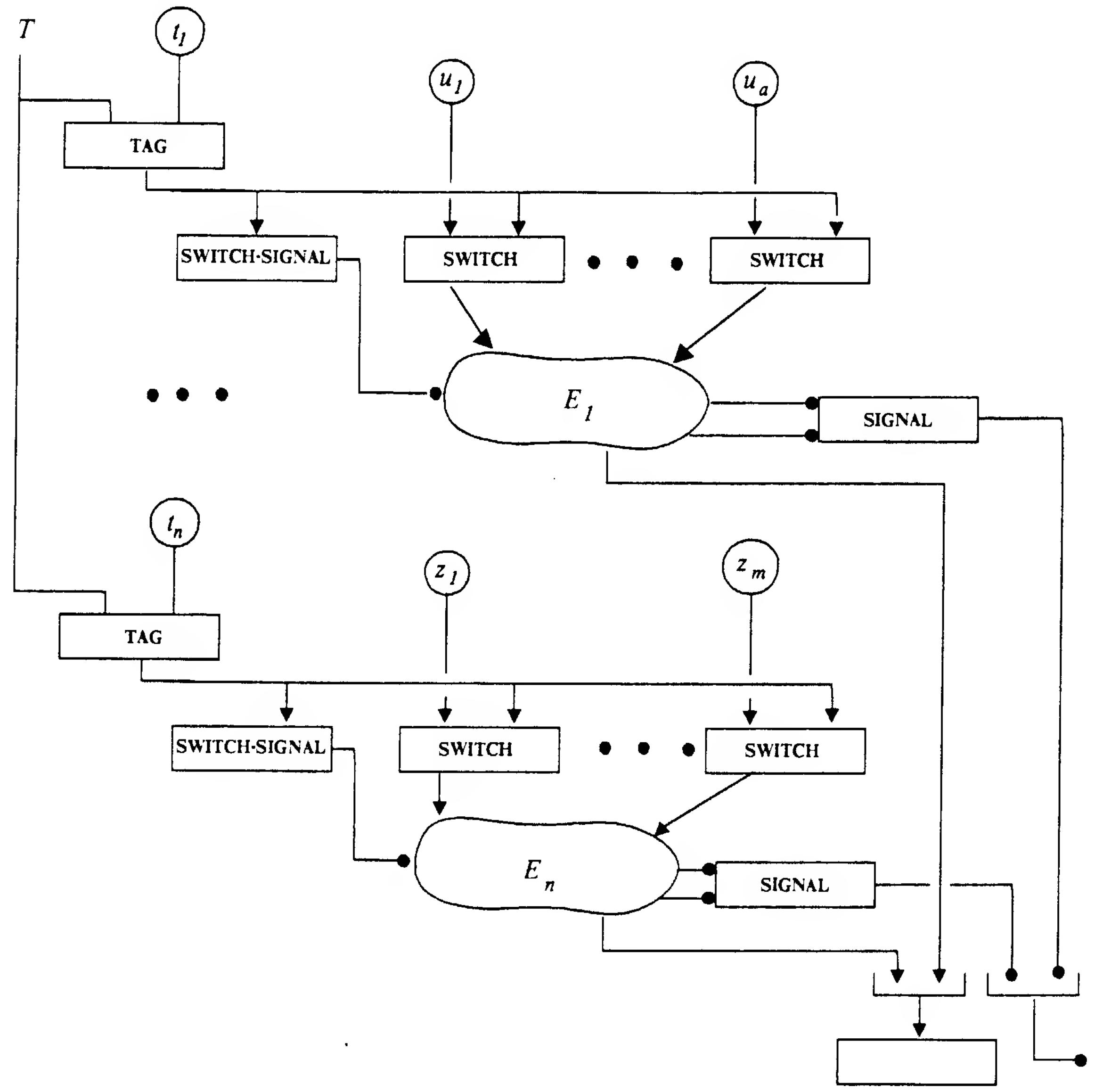


Figure 20: Translation of a tagcase expression

5.4 Function Application and Returns

The syntax for function application in VIMVAL is :

$f(x_1, x_2, ..., x_n)$

where f is the name of a function. In most cases, the function application is performed by the APPLY operator. The arguments x_1, ..., x_n are collected into a single record all of whose elements are intially EC-elements. The code is shown in figure 22. The TAILAPPLY instruction is used in place of the APPLY whenever the compiler can recognize that a function application is tail-recursive. The instruction STREAM-TAILAPPLY is used only in the body of stream producers and will be discussed in the next section.

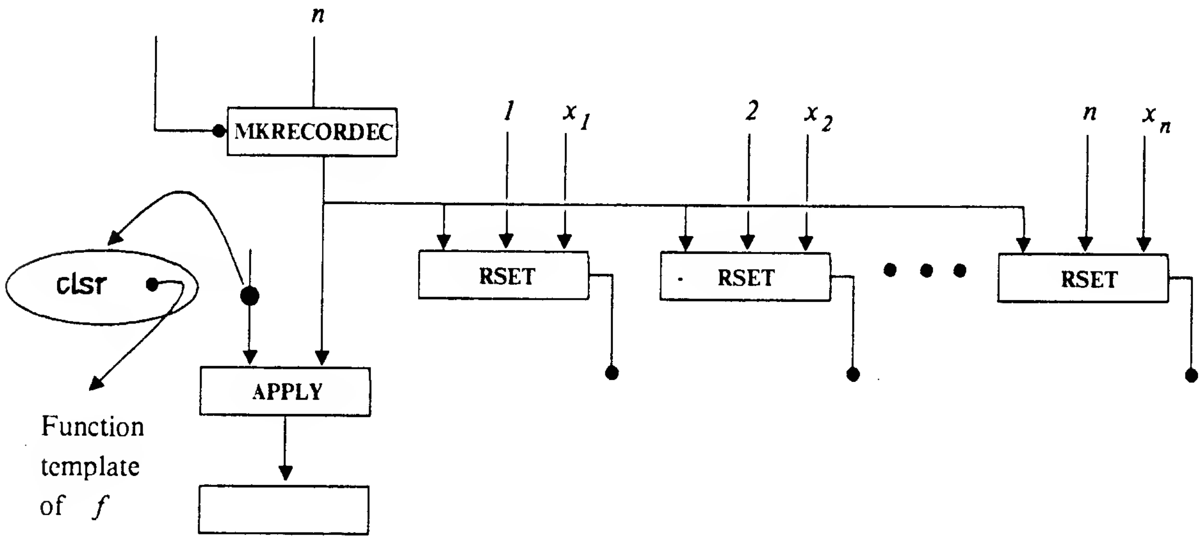


Figure 21: Code for creating the argument record for function activation.

The RETURN instruction is used to dispatch the results computed by the function to the destinations listed in the return link, which is its first operand. The return-link was received by the template from the APPLY instruction. The results computed by the activation are packed into a record, the fields of which are EC-elements. Values returned from a function are similarly packed into a record. In most cases, one of the destinations for the signal produced by the RETURN instruction is the RELEASE instruction for the activation.

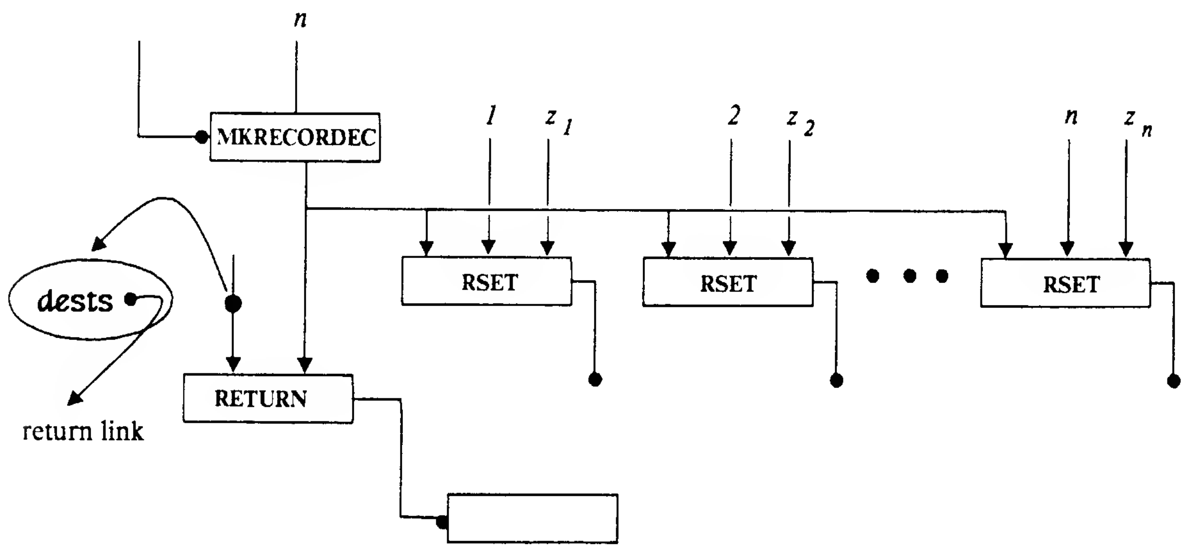


Figure 22: Code for returning the result of an activation.

5.5 Stream Producers and Consumers

A stream is a sequence of values, all of the same type, that are passed in succession, one-at-a-time between functions. The operations on values of type **stream** of type *T* are defined below where *S* and *S'* are streams, and *v* is a value of type *T*.

1. [][*T*] : returns an empty stream (of elements of type *T*) which is the sequence of length zero.

2. **first**(*S*) : The result is the value *v* which is the first element of the stream *S*. If *S* = [] (the empty stream), then the result is *undef*.

3. **rest**(*S*) : The result is the stream left after removing the first element of *S*. If *S* = [] the result is *undef*.

4. **affix**(*v*, *S*) : The result is the stream *S'* whose first element is *v* and whose remaining elements are the stream *S*.

5. **empty**(*S*) : The result is *true* if *S* = [], *false* otherwise.

For a non-empty stream S, the following property is satisfied :

$$S = \text{affix}(\text{first}(S), \text{rest}(S))$$

In VIM the storage representation for a stream is a chain of oneofs. Operations on streams are expressed in terms of operations on the components of the oneofs. The data structure for a stream whose elements are of type *T* is:

```
stream[T] = oneof[empty: null;
                  nonempty: record[first: T;
                                   second: stream[T]]]
```

The following discussion describes the rules using which the compiler can translate the VIMVAL text into data flow graph. The translation rules specified are by no means complete; only the simpler cases are dealt with in this thesis and the more complex cases need further investigation.

The expression [] for creating an empty stream is translated into an expression for creating a oneof with tag *empty*.

```
make[empty: nil]
```

first(*S*) is tranlated into the following code:

```
tagcase(S)
 tag empty: undef;
 tag nonempty: S.first
endtag
```

rest(*S*) is tranlated into the following code:

```
tagcase(S)
  tag empty: undef;
  tag nonempty: S.rest
endtag
```

The code generated for the **affix**(*v*, *S*) is shown below in figure 24. The instructions are so organized that the computation of the rest of the stream is suspended until some instruction demands it. When some computation attempts to perform the **rest** operation on the resulting stream, the suspension is replaced by a pointer to the next element of the stream and a signal is sent to the instruction which initiates the computation of the next element.

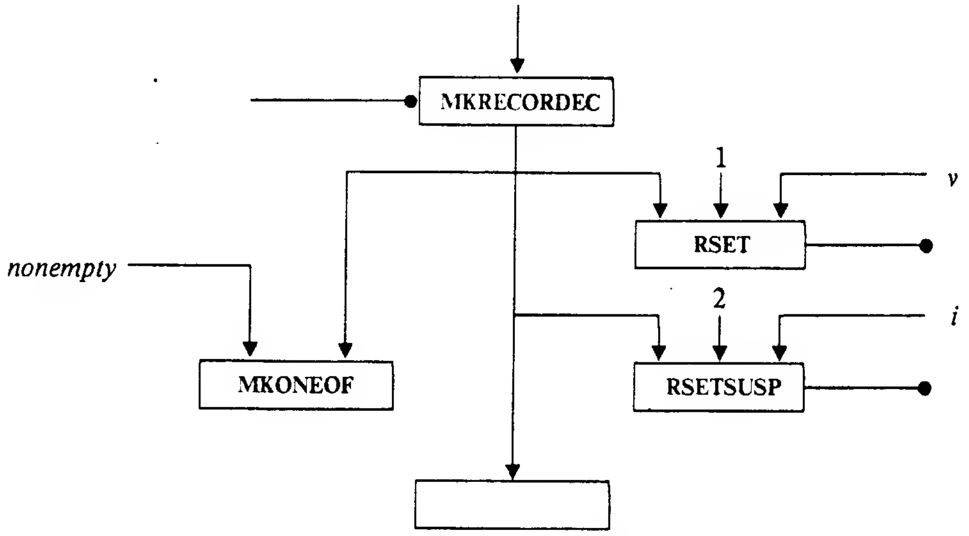


Figure 23: The general form of the base language graph for the expression **affix**(v, S). i is the index of the instruction in the current template which starts the computation of the stream S.

The translation for self-tail-recursive stream producers is quite interesting. I assume that the compiler can recognize stream producers which are self-tail-recursive. The use of tail recursion allows the activation template of the caller to be released before the computation of the callee is completed. This is a significant optimization since it results in a much lower amount of storage that is required for the computation. Mutually tail-recursive programs are translated naively, using simple APPLY and RETURN instructions without taking advantage of the STREAM-TAILAPPLY instruction.

Let f be a self-tail-recursive function that requires n arguments and produces a stream. Let the function be of the form :

function $f(x_1, ..., x_n)$ **returns stream**[T];
body of function
endfun;

The compiler generates an auxiliary function f-aux from f. The code for the body of the function f is generated using the rules specified above, except that every instance of **affix**(v, f(...)) is translated into the graph in figure 25.

The VIMVAL text of f-aux is the same as that of the function f. The only difference is that each

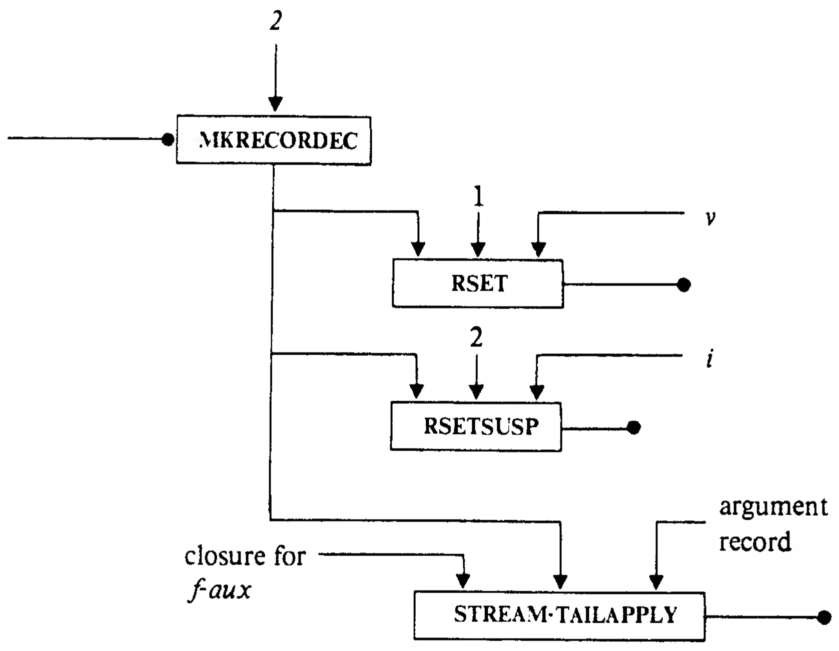


Figure 24: Translation of affix(v, f(...)) in the body of the function f.

instance of affix(v, f(...)) in f is translated into the graph shown in figure 26.

The correctness and generality of these translations are under current investigation and will be the subject of another treatise.

5.6 Discussion

We described a base language such that machines L1 and L2 are equivalent for all programs written in this language. Stated in another way, programs in this base language ensure that when a program halts, the only elements on the heap are those that represent the result value of the computation. Since the data structures are stored in chunks, this ensures that all chunks which were acquired during the computation and which are not part of the result structure are reclaimed.

The base language uses early-completion elements for creating argument lists for function invocations and for creating records in which the result of a function invocation is returned. The use of early-completion elements in these records allows a function to be invoked even if all the arguments

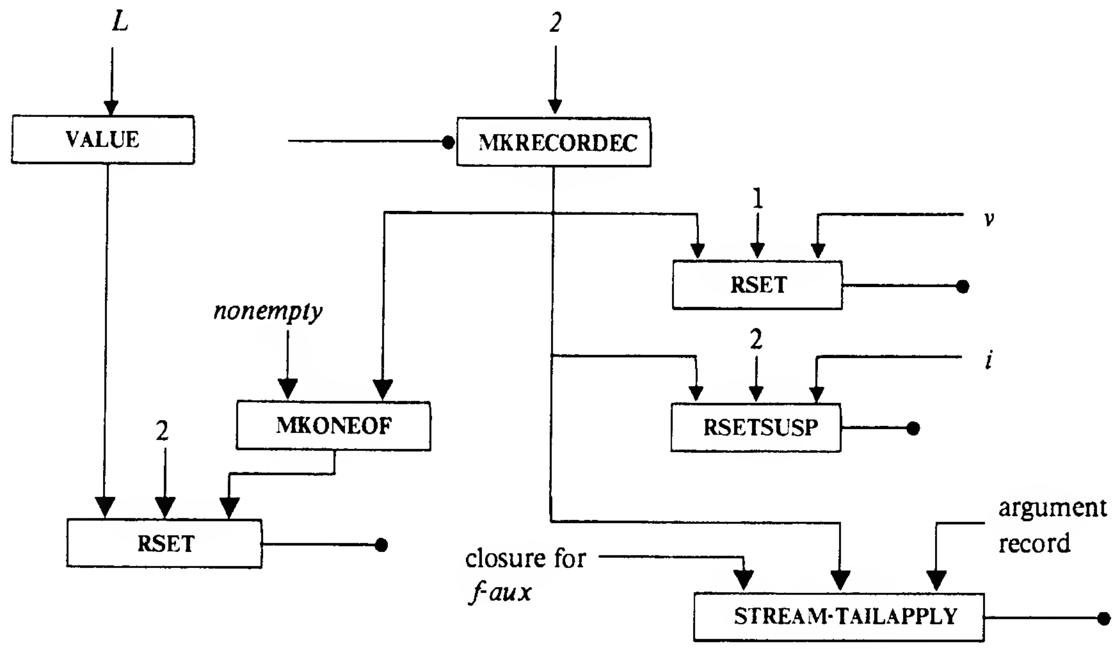


Figure 25: A base language graph for the expression affix(v, $f(x_1, x_2, ..., x_n)$) in the body of f, which is self-tail-recursive, is translated into the above graph in the body of *f-aux*.

have not been evaluated. Similarly, early-completion queues in return records allows values to be returned to the caller even if the computation of all the values which are to be returned has not completed. The other use of the early-completion structures is in the construction of streams.

A major use of the early-completion feature of the language is in the construction of arrays. Creation of arrays whose elements are then initialized by large computations can benefit from the use of EC-queues. Such techniques for increasing the amount of parallelism in programs are the subject of ongoing research.

Chapter Six

Conclusion and Scope for Further Work

The objective of the thesis was to develop a storage management strategy for VIM. An abstract architecture for VIM was informally discussed and some of the distinctive features of VIM explained in an informal manner. This was followed by a formal model L1 of the abstract architecture. The thesis then went on to refine the model L1 to include hierarchical storage consisting of main memory and disk. Chunks, which are the unit of storage allocation and reclamation of storage and the unit of data transfer between main store and disk, were used as the constituent of a new data structure called a VIM-tree. VIM-trees are used to represent structure values (arrays, records and oneofs). An automatic storage reclamation strategy was developed using reference counting. Particular attention was paid to ensure that the machine L2 exhibited desirable behaviour in the presence of EC-queues and suspensions.

A concurrent objective of the thesis was to demonstrate the usefulness of the methodology of computer design by successive refinement. We started with an abstract machine and developed a formal specification for it. The machine model was then refined to include a storage model. In order to show that the refined model L2 (with hierarchical memory, paging, dormant instructions and tree structures for storing arrays, etc.) exhibited the same behaviour as L1 for programs written in the base language discussed in the thesis, we proceeded to prove the equivalence of the two machines. A modification of the McGowan mapping was used to accomplish this.

L2 represents a machine which is closer to an envisioned implementation. L2 may now be refined so that EC-queues, **Function** templates and activation templates would also be represented as data structures. This new model, say L3, may then be proved to be equivalent to L2, and hence to L1, using the kind of technique described in this thesis. Such successive refinement would finally yield a machine model which can be directly implemented to **construct** a real machine.

Future Research

There are a number of topics which are natural extensions of the ideas and issues addressed in this thesis.

1. An implementation of VIM : The abstract architecture may be successively refined to produce a model which reflects the characteristics of the physical elements of a machine – disk interaction, paging algorithms, process priorities, non-terminating computations, faults and exception handling, etc. Each model must be shown to be equivalent to the preceding model, and thus the final implementation would satisfy the specifications of L1 and the two would be computationally equivalent. It is a matter of conjecture as to how far this process of refinement can be performed before the designer is overwhelmed by the details of the machine formalism.

2. Storage Management and Guardians : Guardians are a special **construct** proposed in VIMVAL [13] which allows the programmer to express indeterminacy in computations. They are similar to the **manager** construct in Id [3] and allow the programmer to write programs for, say, data base transactions. It remains to be investigated how the incorporation of guardians in the abstract model would affect the reference counting scheme.

3. Storage Management in Multiprocessors : It would be interesting to develop a model of VIM which has multiple processors and prove the equivalence of this model to L1 for the base language in consideration in this thesis. The issues of storage allocation and reclamation and instruction scheduling can be formally addressed in this model.

4. Extending the Base Language : The base language presented in this thesis is being extended to express efficient computations on arrays. Judicious use of EC-queues should significantly increase the amount of concurrency in the program. This increase in parallelism can be exploited to overlap disk activities in a single processor implementation of VIM, or by multiple processors. VIMVAL constructs which corespond to these new base language constructs must be developed; preliminary research shows that naive extensions of VIMVAL introduces in the type system of VIMVAL.

References

1. Ackerman, W. B. and Dennis, J. B. VAL -- A Value - Oriented Algorithmic Language: Preliminary Reference Manual. 218, Laboratory for Computer Science, MIT, Cambridge, Mass., December, 1978.

2. Arvind, and Gostelow, K. P. "The U-interpreter". *COMPUTER 15*, 2 (Feburary 1982), 42-49.

3. Arvind and J. D. Brock. *Lecture Notes in Computer Science.* Volume 143: Streams and Managers. In *Operating Systems Engineering*, M. Maekawa and L. A. Belady, Ed., Springer-Verlag, 1980.

4. Arvind, and R. A. Iannucci. A Critique of Multiprocessing von Neumann Style. Proc. of the 10th International Symposium on Computer Architecture, June, 1983.

5. Arvind, and R. E. Thomas. I-Structures: An Efficient Data Type for Functional Languages. TM-178, Laboratory for Computer Science, MIT, Cambridge, Mass., September, 1980.

6. Berry, Daniel M. Block Structure : Retention or Deletion. Proc. of the Third Annual Symposium on the Theory of Computing, May, 1971.

7. Bobrow, Daniel G. "Managing Reentrant Structures Using Reference Counts". *ACM Transactions of Programming Languages and Systems 2*, 3 (July 1980).

8. Chu, Yaohan. *High-Level Language Computer Architecture.* Academic Press, New York, 1975.

9. Dennis, J. B. Programming Generality, Parallelism and Computer Architecture. Information Processing 68, 1968, pp. 484-492. Also CSG-Memo 30.

10. Dennis, J. B. On the Design and Specification of a Common Base Language. Proceedings of the Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, New York, April, 1971, pp. unknown.

11. Dennis, J. B. *Lecture Notes in Computer Science.* Volume 19: First Version of a Data Flow Procedure Language. In *Programming Symposium: Proceedings, Colloque sur la Programmation*, B. Robinet, Ed., Springer-Verlag, 1974, pp. 362-376.

12. Dennis, J.B., Guang-Rong Gao and K.W. Todd. A Data Flow Supercomputer. Computation Structures Group Memo 213, Laboratory for Computer Science, MIT, Cambridge, Mass., March, 1982.

13. Dennis, J. B. Data Should Not Change : A Model for a Computer System. Group memo.

14. Dennis, J. B., C. K. C. Leung, and D. P. Misunas. A Highly Parallel Processor Using a Data Flow Machine Language. Memo 134-2, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., June, 1980.

15. Even, Shimon. *Graph Algorithms.* Computer Science Press, 1979.

16. Friedman, D. P., and D. S. Wise. CONS Should Not Evaluate its Arguments. In *Automata, Languages, and Programming*, Michaelson and Milner, Eds., unknown, 1976, pp. 257-284.

17. Friedman, D. P., and D. S. Wise. *Lecture Notes in Computer Science.* Volume 70: An Approach to Fair Applicative Multiprogramming. In *Semantics of Concurrent Computation*, G. Kahn, Ed., Springer-Verlag, 1979, pp. 203-225.

18. Halstead, Robert. *Reference Tree Networks : Virtual Machine and Implementation.* Ph.D. Th., Laboratory for Computer Science, MIT, Cambridge, Mass., 1979. MIT/LCS/TR-222.

19. Halstead, Robert H. MultiLisp : A Language for Structured Concurrency. Real Time Systems Group, Massachusetts Institute of Technology, 1984. Submitted to the *ACM Transactions of Programming Languages and Systems.*

20. Henderson, P.. *Functional Programming: Application and Implementation.* Prentice/Hall International, Englewood Cliffs, New Jersey, 1980.

21. Hudak, Paul, and Adrienne Bloss. The Aggregate Update Problem in Functional Programming Systems. Proceedings of the 1985 Conference on the Principles of Programming Languages, 1985.

22. Jones, Anita K., R. J. Chansler, Jr., I. Durham, P. Feiler and K. Schwans. Software management of Cm* - A distributed multiprocessor. Proceedings of the National Computer Conference, 1977.

23. Jones, A. K. and P. Schwartz. "Experience Using Multiprocessor Systems - a Status Report". *ACM Computing Surveys 12*, 2 (June 1980), 121-166.

24. McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. I. T.. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1965.

25. McGowan, C. An Inductive Proof Technique for Interpreter Correctness. Courant Institute Symposium on Formal Semantics of Programming Languages, 1970.

26. McGraw, J. R. Data Flow Computing - The VAL Language. Memo 188, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., January, 1980.

27. McGraw, J. R. "The VAL Language: Description and Analysis". *ACM Transactions on Programming Languages and Systems 4*, 1 (January 1982), 44-82.

28. Moon, David A. Architecture of the Symbolics 3600. Proceedings of the 12th IEEE International Symposium on Computer Architecture, June, 1985.

29. Myers, Glenford J.. *Advances in Computer Architecture.* John Wiley and Sons, 1981.

30. Myers, Eugene W. Efficient Applicative Data Types. Proceedings of the 11th ACM Conference on Principles of Programming Languages, , 1984, pp. 66-75.

31. Patil, Suhas. An Abstract Parallel Processing System. Dept. of Electrical Engineering, MIT, June, 1967.

32. Steele, G.L. and Sussman, G.J. Storage management in a LISP-Based processor. Proc. Caltech Conf. on Very Large Scale Integration, January, 1979, pp. 227-241.

33. Steele, G.L. and Sussman, G.J. "Design of a LISP-Based Microprocessor". *CACM 23*, 11 (November 1980).

34. Swan, R.J., Fuller, S.H., and Siewiorek, D.P. Cm* - A Modular Multiprocessor. Proceedings of the National Computer Conference, 1977.

35. Swan, R.J., Bechtolsheim, A., Lai, Kwok-Woon, and Ousterhout, John. The Implementation of the Cm* Multi-microprocessor. Proceedings of the National Computer Conference, 1977.

36. Symbolics Inc.. *Symbolics 3600 Technical Summary.* Symbolics Inc., 1984.

37. Turner, D. A. "A New Implementation Technique for Applicative Languages". *Software - Practice and Experience 9* (1979), 31-49.

38. Weng, K.-S. An Abstract Implementation for a Generalized Data Flow Language. TR-228, Laboratory for Computer Science, MIT, Cambridge, Mass., 1979.

39. Wulf, W., Levin, A.R., and Harbison, S.. *Hydra/C.mmp: An Experimental Computer System.* McGraw-Hill Book Company, New York, 1981.